Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations | 1. Thesis and Dissertation Collection, all items

2001-03

An architecture and prototype system for automatically processing natural-language statements of policy

Ong, Vanessa L.

http://hdl.handle.net/10945/10848

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun

http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN ARCHITECTURE AND PROTOTYPE SYSTEM FOR AUTOMATICALLY PROCESSING NATURAL-LANGUAGE STATEMENTS OF POLICY

by

Vanessa L. Ong

March 2001

Thesis Advisor:	James Bret Michael
Thesis Co-Advisor:	Neil C. Rowe

Approved for public release; distribution is unlimited

20010612 101

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2001	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Title (Mix case letters) An Architecture and Prototype System for Automatically Processing Natural-language Statements of Policy			**5. FUNDING NUMBERS**
6. AUTHOR(S) Ong, Vanessa L.			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**. PERFORMING ORGANIZATION EPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(maximum 200 words)***

Organizations are policy-driven entities. Policy bases can be very large and complex; these factors are compounded by the dynamic nature of policy evolution. Thus, comprehension of the ramifications of both policy modification and assurance of the consistency, completeness, and correctness of a policy base necessarily requires some level of computer-based support.

A policy workbench is an integrated set of computer-based tools for developing, reasoning about, and maintaining policy. A workbench takes as input a computationally equivalent form of policy statements.

In this thesis we explore approaches for translating natural-language policy statements into their equivalent computational form with minimal user interaction. We present the architecture of a natural-language input-processing tool (NLIPT), which we designed to augment a policy workbench. NLIPT components consist of an extractor, index-term generator, structural modeler, and logic modeler.

We experimented with a prototype of the extractor. The extractor successfully parsed twenty-seven of a sample of ninety-nine of U.S. Department of Defense security policy statements. An additional twenty-one statements were correctly parsed based on the syntactic structure of the input.

14. SUBJECT TERMS Natural-language Processing, Policy, Security, Formal Methods			**15. NUMBER OF PAGES** 108
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

AN ARCHITECTURE AND PROTOTYPE SYSTEM FOR AUTOMATICALLY PROCESSING NATURAL-LANGUAGE STATEMENTS OF POLICY

Vanessa L. Ong
Lieutenant, United States Naval Reserve
B.S., University of Oklahoma, 1990

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2001

Author: Vanessa L. Ong

Approved by: James Bret Michael, Thesis Advisor

Neil C. Rowe, Co-Advisor

Dan C. Boger, Chairman
Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Organizations are policy-driven entities. Policy bases can be very large and complex; these factors are compounded by the dynamic nature of policy evolution. Thus, comprehension of the ramifications of both policy modification and assurance of the consistency, completeness, and correctness of a policy base necessarily requires some level of computer-based support.

A policy workbench is an integrated set of computer-based tools for developing, reasoning about, and maintaining policy. A workbench takes as input a computationally equivalent form of policy statements.

In this thesis we explore approaches for translating natural-language policy statements into their equivalent computational form with minimal user interaction. We present the architecture of a natural-language input-processing tool (NLIPT), which we designed to augment a policy workbench. NLIPT components consist of an extractor, index-term generator, structural modeler, and logic modeler.

We experimented with a prototype of the extractor. The extractor successfully parsed twenty-seven of a sample of ninety-nine of U.S. Department of Defense security policy statements. An additional twenty-one statements were correctly parsed based on the syntactic structure of the input.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION 1

II. A POLICY WORKBENCH 5
- A. POLICY 5
- B. POLICY WORKBENCH 6
- C. A GENERAL POLICY WORKBENCH 7

III. NATURAL-LANGUAGE INPUT PROCESSING TOOL 13
- A. INTRODUCTION 13
- B. POLICY WORKBENCH ARCHITECTURE 14
- C. NLIPT ARCHITECTURE 16
  - 1. Extractor 17
  - 2. Index-Term Generator 18
  - 3. Structural Modeler 19
  - 4. Logic Modeler 20
- D. SUMMARY 21

IV. RELATED WORK 23
- A. CONTEXT FREE GRAMMARS AND MODAL LOGIC 23
  - 1. Context-Free Grammars (CFG) 23
  - 2. Modal Logic 25
- B. THE BRITISH NATIONALITY ACT AS A LOGIC PROGRAM 25
- C. INCAS: A LEGAL EXPERT SYSTEM FOR CONTRACT TERMS IN ELECTRONIC COMMERCE 27
- D. SACD: A SYSTEM FOR ACQUIRING KNOWLEDGE FROM REGULATORY TEXTS 29
- E. PARALLEL NATURAL-LANGUAGE PROCESSING ON A SEMANTIC NETWORK ARRAY PROCESSOR 31
- F. KNOWLEDGE EXTRACTION FROM TEXT: MACHINE LEARNING FOR TEXT-TO-RULE TRANSLATION 33
- G. UNDERSTANDING OF TECHNICAL CAPTIONS VIA STATISTICAL PARSING 36

V. EXTRACTOR COMPONENT OF NLIPT 39
- A. INTRODUCTION 39
- B. OVERVIEW 39
- C. EXTRACTOR PROGRAM 41
- D. SAMPLE OUTPUT FROM EXTRACTOR COMPONENT 43

VI. TESTING THE EXTRACTOR COMPONENT 47
- A. OBSERVATIONS 48

VII. CONCLUSIONS AND FUTURE WORK 51
- A. CONCLUSIONS 51

B. FUTURE WORK 52
APPENDIX A: EXTRACTOR CODE 55
APPENDIX B: PENN TREEBANK TAG-SET (CONDENSED) 79
APPENDIX C: SAMPLE OUTPUT FROM THE EXTRACTOR 81
LIST OF REFERENCES 83
INITIAL DISTRIBUTION LIST 87

LIST OF FIGURES

Figure 1: Relationship between the Policy Workbench Tools. After [SIBL92]. 10
Figure 2: Natural-language Input Processing Tool is an integral part of the Policy Workbench. 14
Figure 3: Policy Workbench with Natural-language Input Processing Tool. 16
Figure 4: Proposed Architecture of Natural-language Input Processing Tool. 17
Figure 5: Data flow diagram of the Extractor Component of NLIPT. 41

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1: Results from Testing the Extractor Component.....48

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I gratefully acknowledge and thank both Professor Rowe and Professor Michael for their guidance, support, and vision. I thank the Language Technology Group for granting an academic research license to evaluate the LT CHUNK system. I also thank Dr. Moulin, who graciously provided copies of his published papers - twice. Finally, I thank my parents, Veronica and William Thorpe, whose love and support have made all things possible.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Organizations are policy-driven entities. Policy bases can be very large and complex; these factors are compounded by the dynamic nature of policy evolution. Thus, comprehension of the ramifications of both policy modification and assurance of the consistency, completeness, and correctness of a policy base necessarily requires some level of computer-based support.

A policy workbench is an integrated set of computer-based tools for developing, reasoning about, and maintaining policy. A workbench takes as input a computationally equivalent form of policy statements.

In this thesis we develop a system that maps natural-language policy statements to an equivalent computational from with minimal user interaction. We propose the architecture of a natural-language input-processing tool (NLIPT), which we designed to augment a policy workbench. The primary components of the NLIPT are the following: an extractor, which generates a meaning list representative of the natural-language input; an index-term generator, which identifies the key terms used to index relevant policy schema in the policy base; a structural modeler, which structures a schema for input; and a logic modeler, which maps the schema to an equivalent logical form.

We experimented with a prototype of the extractor. The extractor successfully parsed twenty-seven of a sample of ninety-nine of U.S. Department of Defense security policy statements. An additional twenty-one statements were correctly parsed based on the syntactic structure of the input.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Organizations are policy-driven entities. Promulgation of policies to those expected to adhere to them can be fastidious or lackadaisical. Nonetheless, organizations expect their members to adhere to both explicit and implicit (unwritten) policies. Adherence to an organization's policies can be difficult, especially when the policy base is large or there is much implicit policy. Moreover, complex relationships among policies and conflicting policies can lead to errors in the interpretation, refinement (i.e., implementation of policy as procedures in information systems) and enforcement of policy.

Ideally, an organization's policies would be stored in a computational form in a central repository. Users could search the repository for policies that are applicable to a given action or plan (i.e., a sequence of actions for reaching a goal state) within a specific context. Queries to the repository would be through an interface. In addition, authorized users could update the policy base to reflect changes in the organization's policy. Such an interface could be part of a larger system that we term a "policy workbench." A workbench is a suite of tools that serves as an expert database management system. A policy workbench could update policy and test the policy for gaps; by "gap" we refer to any type of error in policy, its refinement, or implementation. It could also map the policy to procedures, which are the mechanisms that implement policy. Thus, a policy workbench is intended to enable the user to represent, reason about, maintain, implement, and enforce policy [SIBL92].

A policy workbench could assist members of an organization to better understand and become more aware of policy, which is necessary for acting in a manner that conforms to policy. Usability of the interface is important. People are less likely to use a system that requires cumbersome structured input, no matter how spectacular its results. An alternative would be to interact with the workbench using a natural language. However, efficient automated processing necessitates converting the natural language into a computational form, usually expressible as well-formed formulae in a formal language [MICH93].

In this thesis we examine processes that map policies submitted in a natural language to formats suitable for further processing by a policy workbench. The applicability and extensibility of various approaches proposed within the natural-language processing community are explored. Correct semantic interpretation of the input is also important. Errors in interpreting policy could become embedded in the computational model of the policy making the resulting model of dubious value for use by the tools. Processing inputs submitted in a natural language entails the following: semantic interpretation of submitted input; mapping the interpretation to an equivalent computational form; identifying applicable existing policies; and submitting everything to the appropriate workbench tools for further processing such as consistency checking. The scope of this thesis is limited to the first two processes.

The organization of this thesis is as follows. Chapter II provides an overview of the policy workbench proposed in [SIBL92]. Chapter III explores issues to be addressed in developing the input-processing component. Chapter IV summarizes some of the previous research that is closely related to this thesis; we discuss the extensibility and

applicability of each approach to processing natural-language statements of policy, in the context of the overall operation of the policy workbench. Chapter V chronicles the development of part of an automated process for converting policy to an appropriate computational format. Chapter VI summarizes the results of testing the automated tool presented in Chapter V. Chapter VII presents conclusions and recommendations for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

II. A POLICY WORKBENCH

This chapter provides an overview of policy and policy workbenches.

A. POLICY

Random House Unabridged Dictionary [RAND93] defines *policy* as follows:

> *1. A definite course of action adopted for the sake of expediency, facility, etc.; 2. A course of action adopted and pursued by a government, ruler, political party, etc.*

Policy serves as a guide in decision-making processes. It can exist either explicitly or implicitly. Explicit policies are deliberately articulated. Implicit policies can arise from traditionally accepted and expected behaviors within an organization; they can also evolve to address gaps in explicit policies.

Policies typically exist in a hierarchy, progressing from broad-spectrum policies at the top to more narrowly defined policies at the lower levels. Policies have both a domain and scope [MICH93A]. The domain specifies the objects in the organization's environment. For instance, policy pertaining to computer security might include system administrators, user accounts, and passwords as part of the domain. Scope identifies the range of roles, obligations, and rights of objects within its domain. As an example, the scope of a system administrator's role might include review of audit trails but not procurement of new equipment.

There are many different types of policy. In this thesis we distinguish among meta-policy, goal-oriented policy, and operational policy. Meta-policy is policy about policy [MICH91]; meta-structures can prove useful in indexing heterogeneous systems. An example of a meta-policy is "*Any policy related to system access is a security policy.*"

Goal-oriented policy states the desired outcome but gives little or no indication of how to obtain the outcome [MICH91]. An example of goal-oriented policy is "*Passwords must be difficult to guess.*" Operational policy defines required actions but rarely identifies the goal [MICH91]. An example of an operational policy is "*Passwords shall be changed every six months.*"

There has been much research in the area of formal representation of policy. Ambiguities in natural-language statements used to represent policy can lead to several interpretations of the policy. Formal representation of policy can, to some extent, remove the ambiguity. Formally represented policies can be defined by axioms and reasoned about using automated systems [MICH91, CHOV91]. Three particular properties of policy are of significance when translating it to a formal representation: completeness, consistency, and correctness. Completeness means that the entire policy base is represented. Consistency means that contradictions within the policy base do not exist. Correctness means that the representation of the policy actually conforms to the real-world intent [SIBL92].

B. POLICY WORKBENCH

A policy workbench is an automated knowledge-based system comprised of a suite of tools designed to assist the user in the representation of policy; reasoning about the properties of policy such as consistency, completeness, soundness, and correctness; refinement of policy; maintenance of policy; and possibly enforcement of policy. A policy workbench can provide several functions depending on the implementation. Ultimately, a policy workbench's purpose is to facilitate adherence to policy.

Use of automation to maintain, reason about, refine, or enforce policy has been examined in several projects. In 1982 ZOG, a menu-based display system developed at Carnegie-Mellon University, was installed on the aircraft carrier USS CARL VINSON. It served as an aid in information management and decision-making in combat situations. Aspects of the ship's tasks, including policy and knowledge from subject-matter experts, were elicited and represented in the knowledge base. [SLOA91]

Regulating Internet and network traffic policies has provided the impetus for many commercial-off-the-shelf (COTS) middleware releases. Although not necessarily a policy workbench, much of this middleware (such as intrusion-detection devices and firewalls) allows a network administrator to select predefined policies or generate their own to enforce an organization's network traffic policies.

The Internet Engineering Task Force (IETF) is developing a policy management architecture that will allow consistent recognition and enforcement of policy protocols. This includes a central policy repository, a common policy definition language, and a common policy object model. The goal is to allow consistent interpretation of protocol policy regardless of the device [STRA99]. Formal languages, such as Ponder [DAMI01] and Path-based Policy Language (PPL) [STON00], have been developed to specify policy about the management of networks and distributed systems.

C. A GENERAL POLICY WORKBENCH

Sibley, Michael, and Wexelblat propose a architecture for a generic policy workbench in [SIBL92].

The authors identify five user classes that should be accounted for in the design of a policy workbench:

1) The policy maker enters policy, maintains a current resource dictionary, confirms consistency of proposed policy statements, allows users to propose scenarios for feedback, and partitions policies into subsets as applicable and necessary.

2) The policy maintainer performs regression testing[1] to ascertain the consequences of modifying policy. It distributes modified policies, in addition to performing configuration management and control tasks.

3) The policy implementer translates policy into procedures, maintains records of rule applications, and maintains a current account of relationships or linkages among policies.

4) The policy enforcer identifies violations of policies and recommends appropriate responses, checks procedures for consistency with policy, and provides authorizations for exceptions to policies.

5) The policy user analyzes the existing policy base via queries.

Figure 1 shows the authors' policy workbench architecture. [SIBL92] described three tools of the workbench:

1) "A *theorem and assertion analyzer* (entering and exercising policy) to check inputs stated as axioms and theorems."

[1] The term "regression testing," as used here, means testing changes in policy from a baseline against some criteria such as correctness or consistency.

2) "A *rule compiler-generator-interpreter* (selecting, merging and generating parts of systems) to produce an executable component of the system[2]." An example of an executable component is a procedure that models a proposed policy environment and provides a run-time scenario for user queries.

3) "An interactive *policy structurer and selector* (aiding in understanding and applying policy) to check what rules are applicable to a given situation and preprocessing the rules into pre- and post-conditions."

Policy input is accepted in a quasi-natural format that is checked for syntactic correctness, then mapped to a formal rule. The rule is submitted to a theorem prover that performs semantic evaluation of the rule to check consistency with policies in the database and to eliminate duplication. If the rule is acceptable, it is sent to the policy database. Conflicts or errors are reported back to the user for action.

The theorem and assertion analyzer also accepts queries regarding policy statements. Accepted in natural language, queries are first submitted to an extractor and translator module, which converts the query to an appropriate computational form. The translation is then processed in a fashion similar to direct policy input with the exception that the policy database is not updated. Rather, a query response is directed to the user.

---

[2] Although the proposed architecture for the workbench could theoretically support many different data models, the examples of policy given in [SIBL92] are represented as conditional rules.

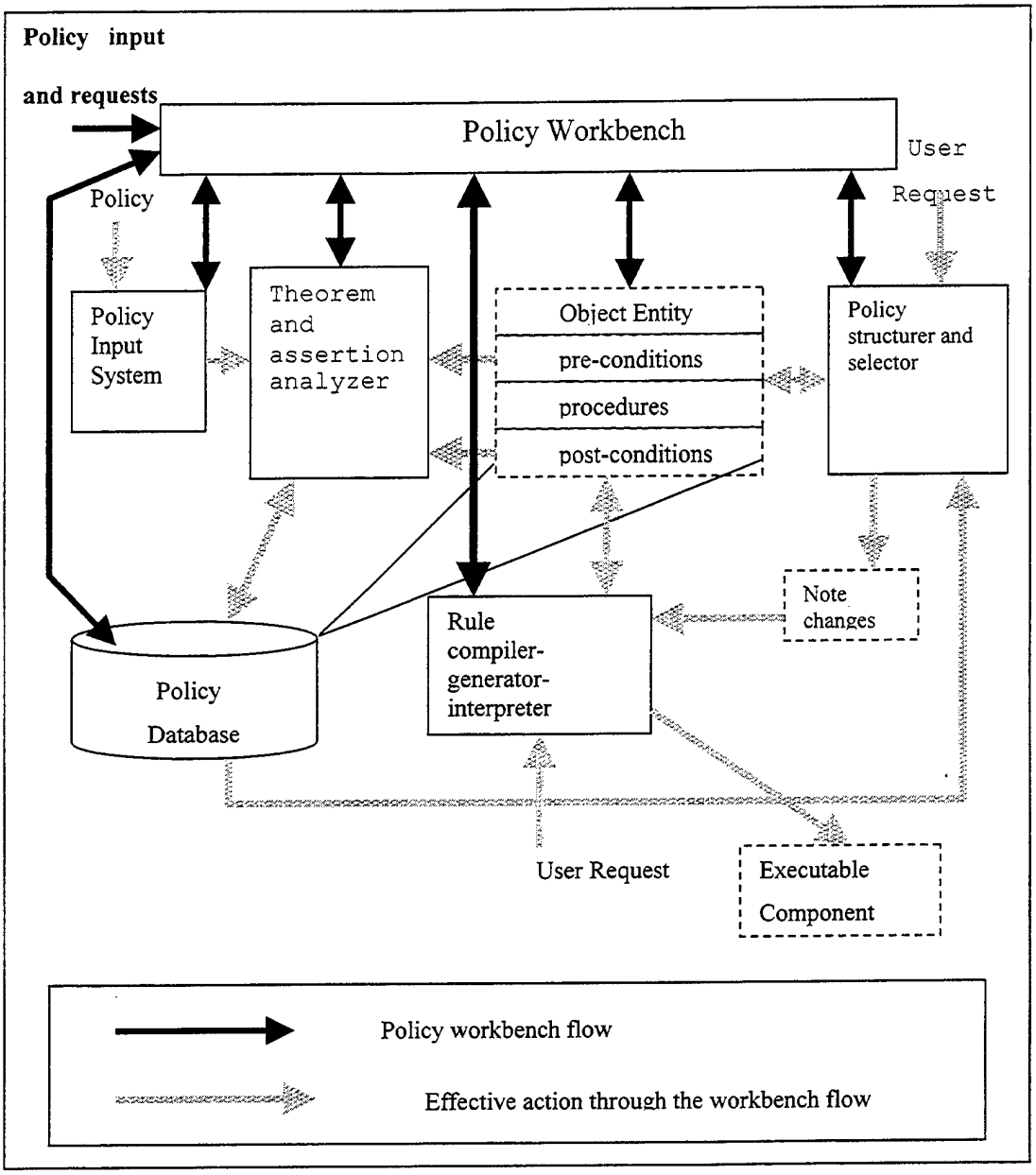

Figure 1: Relationship between the Policy Workbench Tools. After [SIBL92].

The rule compiler-generator-interpreter allows the operation of a simulated system determined by user procedural inputs or scenario requests. Policy changes can essentially be seen "in action."

This policy structurer and selector tool finds policies represented as pre- and post-conditions in the policy database that are applicable to a given input (e.g., scenario or direct policy input). It does so by finding commonalities in policy statements. This information is updated in the resource dictionary. The "understanding module" in conjunction with regression testing would allow the user to discern the effects of policy changes. The understanding module of the policy selector and structurer identifies the relationships between a policy and other components in the database. This tool could also aid in the development of the exceptions required for a policy set.

For this thesis, we will develop the fourth tool of the policy workbench, the policy-input system. We propose to expand the functionality of the policy input system to accept natural-language statements instead of statements in a quasi-natural format. We have renamed this component the natural-language input-processing tool (NLIPT).

THIS PAGE INTENTIONALLY LEFT BLANK

III. NATURAL-LANGUAGE INPUT PROCESSING TOOL

A. INTRODUCTION

A tool within a policy workbench can require as input either update requests or queries about policy. One approach to policy specification is to require that the users of a policy workbench articulate policy in a quasi-natural language[3] "until a more friendly user interface is developed" [SIBL92]. In this approach the policy maker or a formal-methods engineer would be responsible for translating the quasi-natural language statements into a formal language. This expectation, however, may prove to be impractical. The policy base of an organization can be quite large, making the process of manual translation into an acceptable format almost insurmountable. In addition, as policy bases are typically not static entities, frequent updates may be required, placing a further burden on the users. Furthermore, manual translation of policy is an error-prone process, as was demonstrated in experiments reported in [MICH93]. Developing a policy workbench that accepts input and returns output in natural language would greatly relieve the user of repetitive tasks. Otherwise, potential users may balk at using the policy workbench, as they did with ZOG, rendering the workbench ineffective.

We propose that the architecture presented in [SIBL92] be modified to include a natural-language input-processing tool (NLIPT). The NLIPT could consolidate the common input-processing tasks in the workbench. It could extract the meaning from the input and isolate the key components necessary to identify all applicable policies and

[3] We use the term "quasi-natural language" to refer to a language with a restricted vocabulary, syntax, or semantics.

real-world facts (information otherwise unstated in the policy but known to be true and necessary to maintain consistency, completeness, and correctness of the policy base). Most importantly, it could generate the logically equivalent form of the input. The NLIPT should be transparent to the user (Figure 2). In this thesis we do not address the inverse process, that is, translating the computational representation of policy into a natural-language response of the system to the user, such as the answer to a user query about policy.

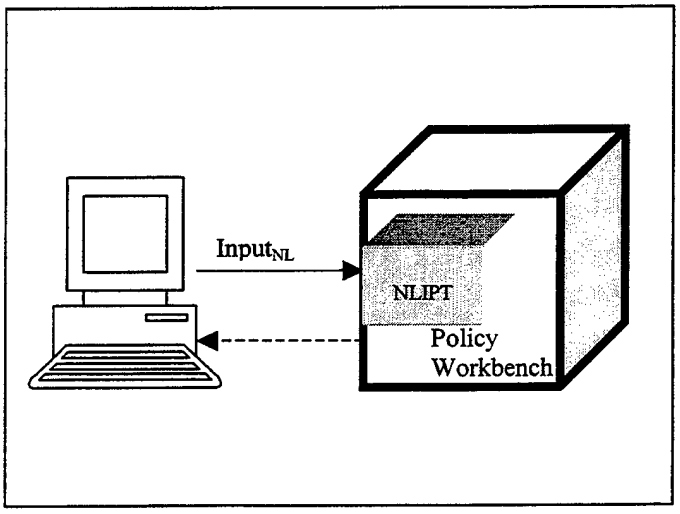

Figure 2: Natural-language Input Processing Tool is an integral part of the Policy Workbench.

B. POLICY WORKBENCH ARCHITECTURE

[MICH93] shows object-oriented modeling of policy appears to produce fewer structuring errors than a non-object-oriented approach. The object-oriented approach begins with a schema, a structural model that defines the entities, mechanisms and relationships contained within the policies. [MICH93] proposes deriving the schema from an extended entity-relationship diagram, where labeled arcs signify the relationships between policy objects. The schema controls the axiomatization of the policies [MICH93]. This does require rephrasing of some policy statements to explicitly refer to

constructs of the schema before formalizing the statement; this assures correct linkage within the model. However, using a structured information model to represent policy and real-world facts can produce a more compact and less error-laden representation than an unstructured approach.

The role of the NLIPT in the policy workbench is illustrated in Figure 3. User input is translated into an equivalent computational form (e.g., first-order predicate logic[4]) via a conceptual schema. This is done for all input, be it policy, queries, or scenarios. Key terms of the input are determined and sent to the policy-element identifier tool, which identifies applicable elements (i.e., policy schema) in the policy base. The inability to find any applicable schema could be used as an indicator that an automatic modification should be tried or that an error message should be sent to the user depending on the type of workbench request. The retrieved schema would be used to formulate a schema for the input, which would then be translated to first-order predicate logic. The input schema and computational form are submitted for processing by the appropriate workbench tools.

Figure 3 shows that the policy workbench connects the user to the tools via a user-interface module, which permits the user to select the desired functionality of the workbench (e.g., policy-base modification, scenario generation, searching). An internal handler would also be required to direct the processed input to the appropriate workbench tool (discussed in Chapter III) and to an exception handler if needed.

---

[4] Studies show the advantages of representing policy using first-order logic: reduction of ambiguity and ability to automate reasoning to name two [CHOV86, SERG86, SIBL92, MOUL92, MICH93]

The policy-element identifier is an extension of the policy-selector tool proposed in [SIBL92], which the authors believed could prove instrumental in the input-processing phase. However, the policy-element identifier differs from the policy-selector tool in that it would retrieve the structural schema of the pertinent items, rather than the computational form. The computational form would not be needed to generate a schema of the input.

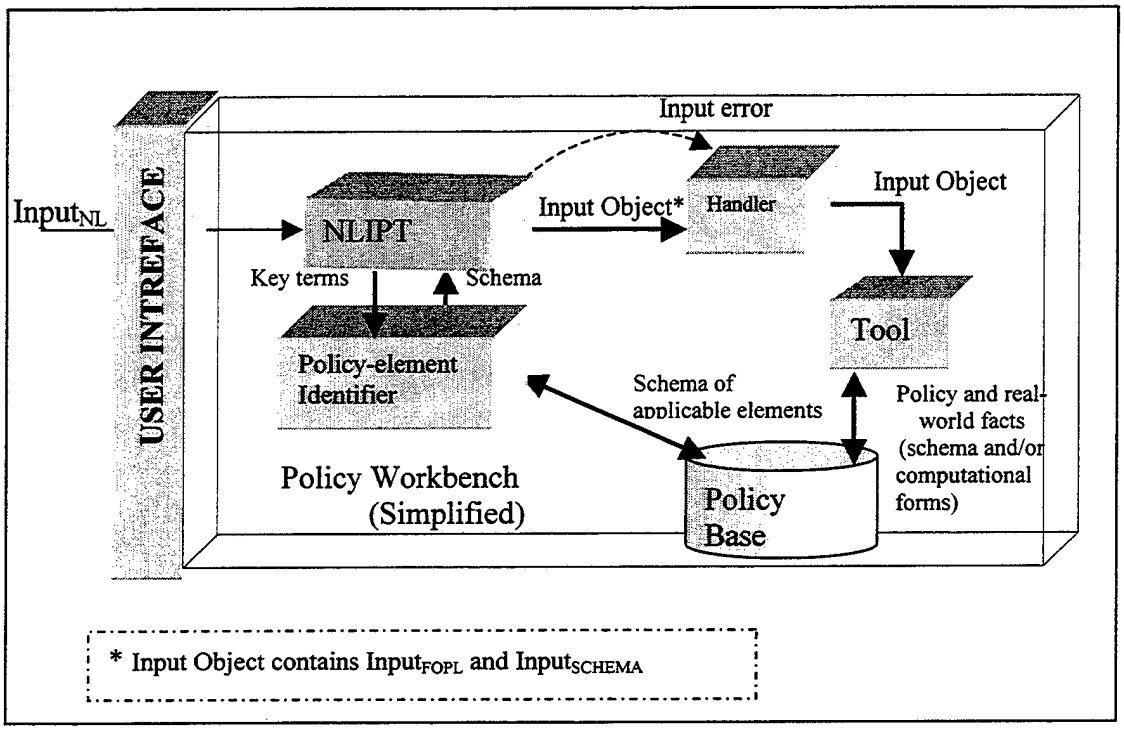

Figure 3: Policy Workbench with Natural-language Input Processing Tool.

C. NLIPT ARCHITECTURE

The proposed architecture of the NLIPT is illustrated in Figure 4. There are four parts: the extractor, which generates a meaning list representing the input; the structural modeler, which generates the schema for the part of the input that is consistent with the schema; the logic modeler, which generates the properly quantified and scoped formal

representation of the input; and the index-term generator, which identifies key concepts of the input. A data dictionary identifies synonyms and probable substitutions for misspelled input.

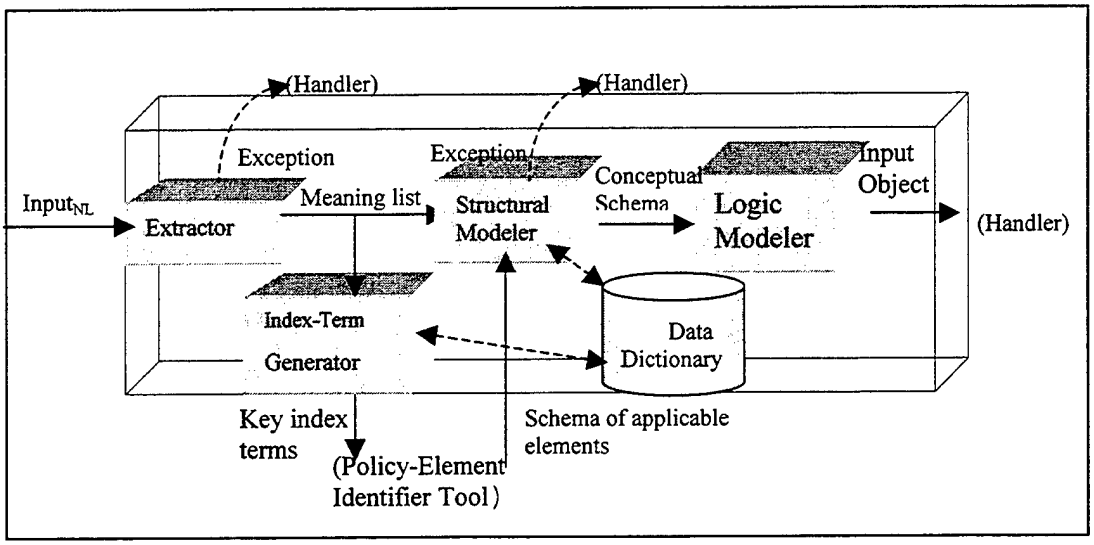

Figure 4: Proposed Architecture of Natural-language Input Processing Tool.

1. Extractor

The extractor generates a meaning from each important word of the natural-language input. At a minimum, the meaning list should identify the subject, object, and attributes of all actions. As an example, consider:

(P1) All passwords must be at least eight characters in length.

Alternatively, it could be stated as follows: *All passwords must contain at least eight characters.* At some point in time the system should be able to recognize that both statements are semantically equivalent. A meaning list for (P1) could be as follows:

ML[subject(passwords),subjectquantifier(all),object(characters), objectmodifer(eight), objectquantifier(at least), subjectattribute(length), action(modal(must), verb(be))][5]

[5] The presented format is not an implementation requirement.

Identifying and correctly attributing modifiers, quantifiers, and conditionals is important. From the example, the extractor should identify the prepositional object "length" as an attribute of password to which "eight" refers. In addition, the adverbial term "at least" signifies that the minimum requirement for the length of a password is provided in that statement.

2. Index-Term Generator

The index-term generator extracts the terms from the meaning list most likely to find a relevant match in the policy base. At a minimum, the subjects, objects, and attributes are tried. Verbs are important as well; for instance, the verb group "must be" signifies that the object is an attribute of the subject. So for the example, the following index terms should be:

subject(passwords), object(characters), attribute(length), verb(be)

The root form of each candidate term is looked up in the lexicon to find synonyms and possible substitutions for morphological variants and misspellings. For example:

Synonyms(password, [password, password ,code word, key, log on, access])
Synonyms(character, [symbol, term])
Synonyms(length, [size, measurement, duration])
VerbSense(be , [property])

The index-term generator must weight the index terms to maximize the likelihood of selecting schema in the policy base that have a high degree of relevance to the input. Subject terms should have the highest weighting; original input should have a greater weighting than synonyms. The synonym term "duration" in the example is not accurate in the context it is used in the sentence; appropriate weighting of synonyms would account for this. Weighting should generally increase with the rarity (i.e., infrequency of

use) of the word; for instance, the term "property" is a commonly used term and likely to generate a number of hits in the policy base if the term is used in a query about policy.

3. Structural Modeler

The structural modeler would analyze any schema retrieved by the policy-element identifier to match the input. The retrieved schema identifies implicit facts and hierarchical relationships. If no applicable schema can be found in the policy base, the structural modeler can either proceed or generate an exception. For query processing or scenario generation, an exception should be noted; for policy assertion, the modeler should continue since the input is a new policy.

Continuing the example, the policy-element identifier should find the following applicable schema:

password(pre_condition(agent: user, agent_status: authorized, action: issued), property(valid), post_condition(agent: user, action: logon))

user(agent: employee, property(issued_password), status: authorized)

logon(pre_condition(agent: user, action: enter, object: password, object_status: valid, purpose: system_access))

access(pre_condition(agent: user, agent_status: authorized, action: logon, action_status: completed(success)), status(granted), post_condition(agent: user, agent_status: authorized, action: use_of_system, action_status: persistent))

This says that a valid password is issued to an authorized user, an employee, to allow the user to logon. The user must logon to obtain access to the system. Access is granted when a valid password is entered to complete a logon. Nothing was retrieved that defines a valid password, but the new policy proposal states that the length must be at least eight characters. Linkages to the term 'valid' should be made to the schema. (These linkages are not available to concurrent users until the policy is actually accepted

into the database.) The structural modeler should infer that passwords have a minimum length and are comprised of characters. This can be done via the lexicon reference through recognition that the adverbial phrase "at least" is a minimum constraint and that the term "characters" is not a unit of measurement but an object. The term "be" allows the connection to composition. Quantifiers must also be inferred and their scope determined. Further, appropriate generalizations should be made to control the overall policy base size and to ensure that the schema is not too specific, which could affect proper indexing especially for related policies. For our example, the generalization is made that length is actually the size of the password. Hence:

password(applies_to: all, property (size (minimum(eight)), composition(characters))

4. Logic Modeler

The logic modeler uses the schema developed by the structural modeler to generate a first-order predicate logic representation of the input. An appropriate logical representation can vary depending on whether the input is for a query or some other type of request [MICH93]; modularizing the structural modeler from the logic modeler allows for separate generation of the schema, independent of the user request.

Like the structural modeler, the logic modeler must determine quantifiers and their scope. Inferences should also be made as appropriate; for our example, the logic modeler should infer that "composition" from the schema signifies a "part_of" relationship between "character" and "password." For a policy assertion, our example should become:

$\forall_X$ (password(X) $\rightarrow$ ($\exists_S$($\forall_C$ (character(C) $\wedge$ part_of(C,X)) $\rightarrow$ member(C,S))) $\rightarrow$ ($\exists_N$ size(S,N) $\wedge$ N$\geq$ 8)))))

D. SUMMARY

The NLIPT should be a general-purpose tool that requires minimal user interaction after the initial setup. Initial setup should include modification of the data dictionary to accommodate domain specific words and synonyms. The ultimate goal for the NLIPT is to fully automate formal policy representation; while total automation may not be possible, we believe that a good approximation can be made. To demonstrate this, we will implement the proposed NLIPT components.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. RELATED WORK

In this chapter we highlight the work of eight groups of researchers. We provide our observations about the applicability of each group's findings to the policy workbench, and in particular, the NLIPT component of the workbench. We begin, however, with a brief background on grammars and modal logic.

A. CONTEXT FREE GRAMMARS AND MODAL LOGIC

1. Context-Free Grammars (CFG)

A grammar is a finite set of productions (i.e., rules) and symbols used to generate strings that are valid in a language or to analyze the structure of strings ("parse" them). Context-free grammars are used to formalize parsing rules for languages. Natural-language statements can be parsed to identify the key components of the statement (i.e., subject, predicate, and object). A CFG has [HOPC79]:

- a set of terminal symbols (T), which make up the valid strings;
- a set of variables (V), which are a placeholder for sequences of terminal symbols;
- a start symbol (S) that represents the initial string during statement generation; and
- a set of productions (P) that define legal ways to replace a variable in a string by a string of terminal symbols or variables.

As an example, suppose we are given CFG = (V, T, P, S) where V = {<sentence>, <noun phrase>, <verb phrase>, <noun>, <adjective>, <verb>}, T= {dogs, little, bark}, S=<sentence>, and P consists of

<sentence> → <noun phrase> <verb phrase>

<noun phrase> →<adjective><noun>

<noun phrase>→<noun>

<verb phrase>→<verb>

<noun>→dogs

<adjective>→little

<verb>→bark

With this grammar, we can use the first production (since its left side has a matching variable) to replace the start symbol with "<noun phrase><verb phrase>." We can use the second production to replace the variable "<noun phrase>" to get "<adjective><noun><verb phrase>." The fourth production is used to replace "<verb phrase>" to produce "<adjective><noun><verb>." Finally, terminal symbols are used to replace the variables, resulting in the string "little dogs bark."

A CFG cannot provide a complete description of a natural language such as English, but one can come close. Semantic constraints should also be applied to eliminate meaningless strings that are syntactically correct.

2. Modal Logic

A modal is a special marker of verb tenses in English, appearing as an "auxiliary" before the verb, and that often denotes prescriptive information. Examples of modals are "can," "necessarily," "must," and "may". Some modals like "may" or "shall" can convey several meanings, which is of particular concern in policy interpretation. Modal logic includes reasoning about knowledge and the expressions "it is necessary that" and "it is possible that" [STAN1]. Modal logic has evolved, however, to represent a range of related ideas; for instance, deontic logic is concerned with obligation, permission, and interdiction. Modal logic augments first-order logic with modal quantifiers on sentences [RUSS95]. Modal logic allows inferences to be made concerning the knowledge base.

Modal logic is particularly useful in analyzing a policy base because policy is typically prescriptive in nature containing modalities indicating obligations, permissions, and interdictions.

B. THE BRITISH NATIONALITY ACT AS A LOGIC PROGRAM

Sergot, Sadri, Kowalski, Kriwaczek, Hammand, and Cory [SERG86] explore the feasibility of using logic statements to represent part of The British Nationality Act of 1981 and mechanically determine the consequences of the Act when applied to test cases. They showed that formalization of legislation by rules can be used to develop an expert system without requiring much elicitation of knowledge from an expert. The authors list three benefits to be realized through the formalization of regulations: 1) identification and elimination of ambiguity and imprecision; 2) clarification and simplification of the natural-language statement of the regulation; and 3) derivation of logical consequences of the regulations.

The system represents a portion of the Act using extended Horn clauses; the clauses are implemented as a Prolog program using APES, a Prolog-based expert system shell developed by Sergot and Hammond. The collection of clauses is an axiomatic theory, which can be mechanically analyzed by theorem provers; Prolog serves as a limited-purpose theorem prover. The shell queries a user to dynamically supply facts as required.

Sergot and his colleagues followed a top-down, goal-directed manual approach when formalizing the Act. They defined high-level concepts before the lower-level concepts. This allowed them to postpone the representation of lower-level concepts until high-level concepts were refined in their model. They addressed vague concepts such as "good character" by assuming that vague concepts always applied when generating answers; that is, if a person had to be of "good character" to be a citizen, the assumption was that the person had "good character." The authors determined a meaning for ambiguous or imprecise concepts of the Act that could not be addressed by assuming their truth.

Formalized statements of the Act were manually generated and progressively refined on a trial-and-error basis. Modification and restructuring of previously formalized concepts was made as needed when later sections of the Act refined earlier concepts. A closed-world assumption implemented negation as failure (anything which is not known is assumed to be false). Double negation, if treated classically, would cancel out (not [not p] implies p). However, this was not the intent of the Act for all cases. The authors avoid special explicit clauses for every occurrence of double negation

by treating the negative information as part of the input from the user, though they concede that this approach has drawbacks.

The authors also found that counterfactual conditional statements were not adequately handled by extended Horn clause logic, as with the following statement from the Act:

> *...became a British citizen by descent or would have done so but for his having died or ceased to be a citizen...[by] renunciation.*

They addressed the inadequacy by writing additional rules to address the conditionals. This process required a thorough analysis of the provisions of the act and knowledge of the drafter's intent when writing the counterfactual. Addressing counterfactuals substantially increased the overall number of clauses. Finally, discretionary clauses (clauses which give an authority the discretion to modify application of other sections of the Act) were handled by generating two clauses: one for the standard case and one for the discretionary case.

The manual generation of the Horn clauses was an involved task, often requiring revisions. The trial-and-error approach would place a huge burden on policy makers if the policy base were large, so this approach would not scale. Modification or restructuring of rules could easily get out of hand as the number of formalized statements increased. Moreover, the closed-world assumption, while convenient for domains where all cases are specified, would not apply to most real-world policy.

C. INCAS: A LEGAL EXPERT SYSTEM FOR CONTRACT TERMS IN ELECTRONIC COMMERCE

Tan and Thoen [TAN00] developed an automated expert system that provides advice on the use of Incoterms. (Incoterms are thirteen terms used in legal trade contracts

that stipulate which party (i.e., buyer or seller) is responsible for arranging and paying for transport and arranging the required documents for the transport.) INCAS (INCoterms Advise System) is a Prolog-based system that defines Incoterms to the user, reasons using the Incoterms knowledge base to advise on queried scenarios, and proposes the optimal Incoterm for both buyer and seller given their obligations. This system is intended to assist organizations involved in international trade.

INCAS uses formal specification of the Incoterms in Prolog, manually derived from the International Chamber of Commerce (ICC) book *Guide to Incoterms 1990*. A graphical user interface allows the user to view the INCAS response to a query along with the assumptions used to derive the conclusion when applicable. Users can change the assumptions to refine the conclusion and rerun the query. Users can also introduce hypothetical assumptions to generate responses to what-if scenarios.

The Incoterms domain has many instances where defeasible reasoning is involved. Defeasibility means that rules can be superseded by another rule or fact. The authors address defeasibility by incorporating exception predicates into the rules and adopting the closed-world assumption. Exceptions to exceptions are also addressed in a similar fashion.

INCAS performs symbolic processing on strings and does not use any semantic constructs. A user is required to provide the data concerning the situation for which the query has been formulated. It can also accommodate correcting or otherwise modifying assumptions used in deriving a conclusion to generate a new conclusion.

The authors provided no data regarding the difficulty of development and the approach used in the derivation of the predicate statements. Scalability [SERG86] is still a problem since manually formalizing policy is difficult.

D. SACD: A SYSTEM FOR ACQUIRING KNOWLEDGE FROM REGULATORY TEXTS

Moulin and Rousseau [MOUL94] developed a Prolog system named SACD (Système d'Acquisition des Connaissances Déontique) capable of generating a knowledge base from regulatory text by analyzing the text's logical structure. Semantic content is not analyzed by SACD. SACD is specifically designed to work with prescriptive text, especially the normative propositions found in instructional text. Normative propositions are sentences that describe instructions and characteristically contain modal operators. Most regulatory text, such as policies and legal manuals, are prescriptive in nature. The authors use portions of the National Building Code of Canada for analysis.

Regulatory texts contain three types of formats:

1. Definitions (sentences that clarify domain objects). Definitions do not typically contain modals.

2. Normative propositions, propositions containing verb expressions that explicitly indicate obligations, permissions, or restrictions.

3. Meta-textual statements (cross-references to the text structure).

Regulatory text can typically be segregated into three layers:

1. The macrostructure layer, corresponding to headings, titles, chapters, and sections;

2. The microstructure layer, the logical content of the text featuring the expressions that identify conditions, exceptions, modalities, and references; and

3. The domain layer, domain-specific information that belongs to neither of the other two layers.

SACD initially uses context-free macrostructure and microstructure text grammars to parse the input text. The grammars have multiple entry points and behave like chart parsers, the classic bottom-up approach to parsing with a context-free grammar. Macrostructure analysis detects the presentation elements; microstructure analysis uses modal operators (e.g., "may," "must," "cannot"), conjunctions, internal references, and punctuation in order to identify relevant objects in the domain and applicable deontic rules. Deontic rules characterize the modal object to which they refer and require modal logic.

The knowledge base is generated in two phases: (1) for every verb-phrase in the text a deontic rule is generated without considering the internal references (i.e., cross-references to other portions of the same text); (2) for each meta-textual statement encountered, the conditions and exceptions of the rules that are affected by an internal reference are modified to reflect the influence. The knowledge base eventually contains an object-type hierarchy, object descriptions, rule specifications which indicate the modality and characterize the related object, relationships between the data structures,

and the relationships between the structures and the text provisions. Data structures are represented using Prolog predicates.

After each microstructure analysis, the results are presented to the user for acceptance or correction. A "domain specialist" creates the object-domain hierarchy, partly domain-specific, and resolves anaphoric references.

SACD was used on a subset of the National Building Code of Canada (NBC). Of 100 provisions evaluated, the macrostructure analysis took roughly five minutes overall. The microstructure analysis averaged five seconds per sentence depending on the complexity; the total number of sentences was not identified. A simple expert system checked situations against the provisions of the code.

This approach requires well-structured input text. However, many policies are well structured, so this system should work well on them. However, system may not be suitable for handling queries to the policy workbench using natural language. Features of particular use in the policy workbench are its recognition of meta-textual structures and the refinement of rules based on cross-references. SACD requires a lot of user interaction; this makes scalability a concern. Also, the user must have an intimate knowledge of the input text to correctly generate the domain hierarchy, which makes it subject to personal interpretation.

E. PARALLEL NATURAL-LANGUAGE PROCESSING ON A SEMANTIC NETWORK ARRAY PROCESSOR

Minhwa Chung and Dan Moldovan developed a system with a parallel memory-based parser called PARALLEL [CHUN95] that is implemented on a dedicated marker-passing computer called the Semantic Network Array Processor (SNAP). It exploits a

large case memory instead of complex parsing rules and grammars. Parsing is achieved through a marker-passing search that matches input text with template patterns called concept sequences stored in memory in the form of a semantic network of interrelated facts.

Marker-passing is an inference method used to find connections between concepts in a semantic network. Inferences are developed by first propagating markers forward along the superconcept hierarchy from the origin concept and checking for intersections of markers. Next, markers are propagated along the reverse-semantic links. At the end of the processing, inferences can be made about nodes that have both markers. In a parallel implementation, the markers are propagated concurrently to reduce execution time.

A preprocessor applies domain-specific modifications, such as grouping noun groups and expanding contractions, to each input sentence. A phrasal parser groups the relevant words into the following phrasal segments of noun group, verb group, adverb group, date/time group, conjunction, preposition, relative pronoun, punctuation, "that" group, or possessive marker. Concept sequences (i.e., phrasal patterns stored in the knowledge base) are represented as a set of concept-sequence-element nodes attached to concept nodes in the semantic concept hierarchy, arranged in order to match the sequence of input phrases.

The experiments used 500 complex sentences from MUC-4, of which sixty-eight percent were correctly parsed. The authors attributed most of the errors to no appropriate

concept sequences in the knowledge base, unanticipated linguistic phenomena, and unknown input words.

This approach creates a meaning list of the input. It performs the same function as the extractor tool proposed for the policy workbench. It relies on phrasal pattern matching, which makes it somewhat domain-specific. It is also tightly coupled to both a specific hardware platform and system configuration.

F. KNOWLEDGE EXTRACTION FROM TEXT: MACHINE LEARNING FOR TEXT-TO-RULE TRANSLATION

Delannoy, Feng, Matwin, and Szpakowicz [DEL93] investigate natural-language processing to extract knowledge from technical expository texts in the MaLTe (Machine Learning from Text) system. By incorporating both machine learning and natural-language processing, the authors believe they can more thoroughly represent a text than a system using only one of the processes. MaLTe operates with a minimum of a priori knowledge. It extracts from both the narrative text (the authors used part of the personal income tax law as described in *Revenue Canada 1991*) and examples provided. Additional knowledge required to resolve ambiguities and inconsistencies and to define synonyms are elicited from the user as needed.

Facts are generalized automatically from the examples into the higher-level concepts found in the narrative. In doing so, implicit knowledge is made explicit and a hierarchical domain (for the text) is generated. The authors propose absorption, an operator used in inductive logic programming, to achieve this abstraction, but over-generalization is a danger. Related facts obtained from the examples are aggregated. The

constants in the facts are generalized to variables and the aggregation method becomes a rule.

In order to handle texts with nested concepts, explanation-based generalization (EBG) integrates the applicable rules into one that makes the most useful features of the concepts explicit. This will operationalize the rule, that is, make it a procedure. But an acceptable operationality criterion must be determined, and EBG requires a complete domain-knowledge base.

MaLTe is not autonomous; users must supply missing facts, correct mistakes, and address synonyms. The actual extent of the interaction required for a complete knowledge base for some domain is uncertain.

Delannoy and Rios further refined of MaLTe [DELA94] in conjunction with TANKA, a domain-independent, interactive natural-language analyzer. TANKA has two main components: 1) DIPETT, a syntactic parser; and 2) HAIKU, an interactive semantic analyzer using case-based reasoning. Users are required to select which of the proposed relationships is most correct. HAIKU produces a "protonetwork," a collection of syntactic and semantic constructs. MaLTe translates the protonetwork to Horn clauses. The techniques discussed in [DELA93] are applied to the Horn clauses to both generate a domain theory and refine the existing clauses. MaLTe then converts the refined Horn clauses back to protonetwork form, which is submitted to TANKA for assimilation into the semantic network. MaLTe is implemented using Quintus Prolog.

Barker, Delisle, and Szpakowicz [BARK98] developed a metric for evaluating the performance of TANKA. There were three criteria for the evaluation: 1) the ability of

HAIKU to learn to make better proposals to the user measured as the total number of assignments made by the user compared to the total number of correct assignments suggested by HAIKU; 2) the total number of relationships analyzed by HAIKU compared to the actual number in the text; and 3) the burden to the user of having to make a determination of the relationships proposed by HAIKU.

The authors chose a text about small engines to test the system. They drew three main conclusions:

1) TANKA can learn. The system was able to generate correct analyses of inputs it had never seen before by using partial matching on the semantic patterns it had in its knowledge base.

2) Knowledge can be acquired from text with fragmentary parses and even misparses. Imperfect parses do not necessarily result in no of knowledge acquired.

3) The system did not prove to be too onerous for the user. The average user time to determine a correct relationship proposed by HAIKU decreased over the course of the experiment. As the knowledge base grew, the user made fewer corrections to the proposed inferences. The experiments regarding the burden to the user are of note. However, very large bodies of policy may prove onerous to the user if much action is required on every input.

G. UNDERSTANDING OF TECHNICAL CAPTIONS VIA STATISTICAL PARSING

Processing of multimedia captions has some similarities to processing policy statements in having a limited descriptive semantics. The MARIE-2 system [ROWE1] relies extensively on an accurate and domain-specific lexicon stored as Horn clauses and facts. A database containing a full synonym list, an *a-kind-of* hierarchy and a *part-of* hierarchy was created for the domain words. The Wordnet thesaurus system was used to generate most of the information. Implicit lexicon information is generated using special-format rules that recognized patterns for various kinds of code words and abbreviations. The lexicon contains over 21,000 words from Wordnet (6,000 caption words and 15,000 synonyms) and 1700 domain-specific words that required explicit definition. Synonyms for technical word senses were also added. The lexicon is necessarily large to address the technical jargon unique to the domain such as code words, acronyms and unusual words. The domain was technical captions from military photographs.

The system uses a context-free grammar of 192 syntax rules. One hundred sixty rules are binary (involving replacement of one symbol by two); seventy-one of which are context-sensitive. The remaining rules are unary (involving replacement of a symbol by another). The binary rules have associated semantic rules that check semantic consistency, calculate the total probability, and generate the meaning list. Of the 114 associated semantic rules, fourteen are specific to the domain dialect.

A bottom-up chart parser with word-sense statistics determines the most likely interpretation of the input. The word-sense statistics were obtained by extrapolating the

counts of each word from the training corpus. A branch-and-bound search is performed to build up the best phrase interpretations. Ranking uses four factors: (1) word-sense statistics, (2) counts on the grammar rules used, (3) counts on the co-occurrence of pairs of headword senses conjoined in the parse tree, and (4) miscellaneous factors.

This approach also performs the functionality of the Extractor tool proposed in Chapter III while avoiding phrasal pattern matching. This makes it theoretically suitable to all forms of input. However, a representative training corpus must be used to generate the appropriate statistics by forcing the system to backtrack until it obtains the correct parse. This may prove burdensome to the user when initially setting up the knowledge base. This system also requires lexicon information in advance for all words it is to handle.

THIS PAGE INTENTIONALLY LEFT BLANK

V. EXTRACTOR COMPONENT OF NLIPT

A. INTRODUCTION

We now describe a prototype component of a natural-language input-processing tool that we implemented, the extractor. The extractor analyzes the input and generates a meaning list identifying the subjects, objects, attributes (subject and object modifiers), verbs, and modal qualifications.

The extractor component was designed to capitalize on the typical policy structure as described in [MOUL92]. Most policy statements can be segmented into three sections: 1) the main verb group, most likely with a modal qualifier; 2) the *front scope* of the statement containing the subject; and 3) the *back scope* of the statement containing the object of the verb group. Though the program recognizes a typical policy structure, it can also handle statements with different syntactic structures. The intent was to develop a general-purpose component for many policy domains.

B. OVERVIEW

Input policy statements were initially submitted to an English tagger. This significantly reduced the complexity of the extractor program, which used the part-of-speech tags to more easily identify the key items for the meaning list. The tagger used was a syntactic partial parser, LT CHUNK, developed by the Language Technology Group of Edinburgh, United Kingdom [LTG1, LTG2]. Its output assigns a part-of-speech tag to each word or symbol of the input and it brackets key multi-word syntactic units such as noun phrases. LT POS, a component used in LT CHUNK, assigns part-of-

speech tags to words and symbols using hidden Markov models using maximum entropy probability estimators [GROV1]. It contains a tokenizer, a morphological classifier, and a morphological disambiguator [LTG2]. LT POS achieves ninety-six to ninety-eight percent accuracy at correctly assigning POS-tags when all the domain words are in the lexicon [LTG2]. Noun groups and verb groups that it recognizes are also bracketed.

We converted the tagged output via a Java program to a format suitable for manipulation via Prolog. Next a Prolog extractor program generates a meaning list using the tags and basic grammar rules. Figure 5 illustrates the data flow associated with the extractor component. Referring to the design in Chapter III, the output from the extractor component could be submitted to an index-term generator and a structural modeler.

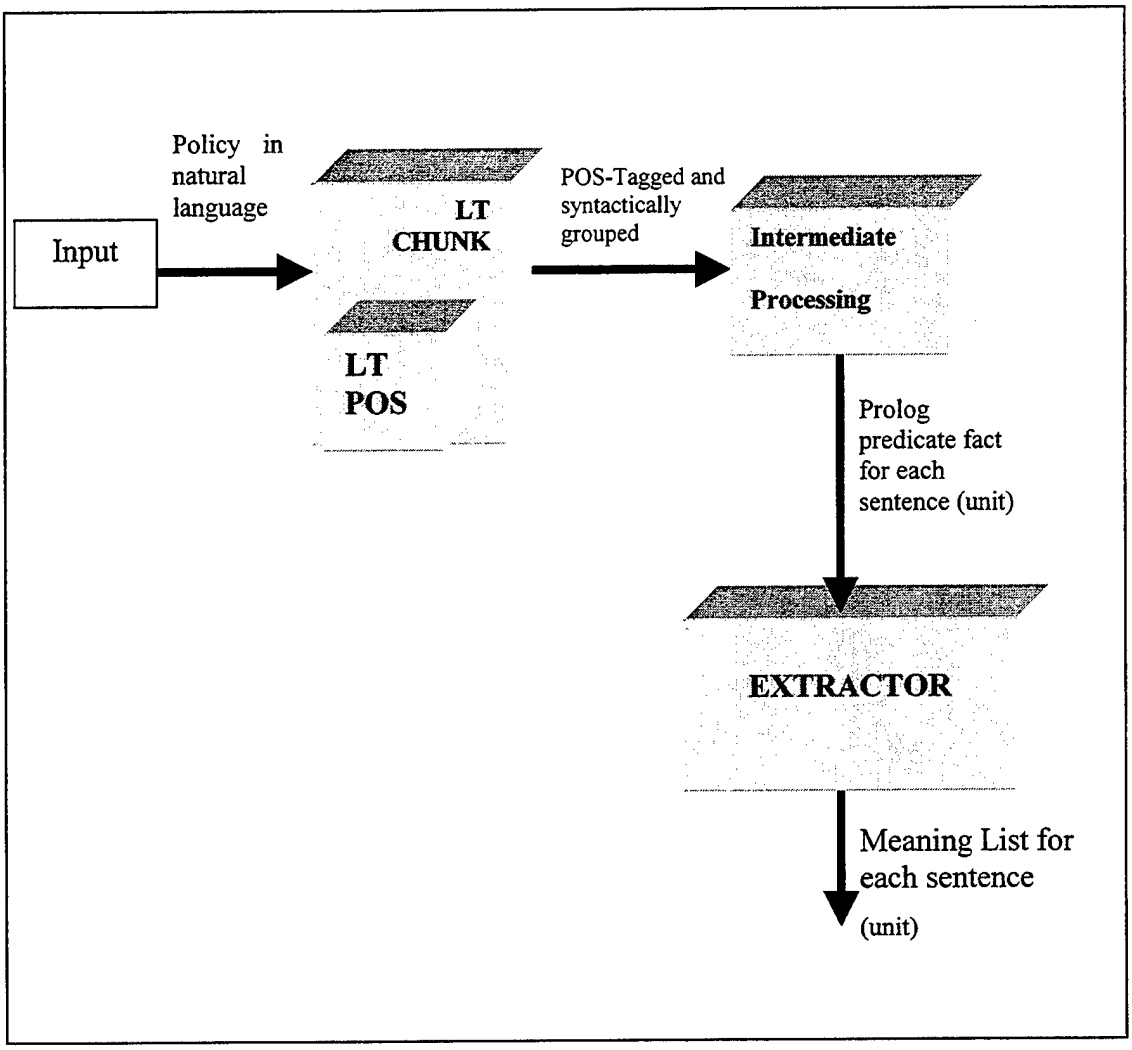

Figure 5: Data flow diagram of the Extractor Component of NLIPT.

C. EXTRACTOR PROGRAM

This program was developed using Prolog. Prolog is advantageous in natural-language processing because of its automatic backtracking feature. If a goal fails, backtracking checks alternative means of satisfying the goals until all possibilities are exhausted [ALLE95]. This is important for natural-language processing because there are usually many ways to interpret a statement in English; parsing may require several attempts before the correct rule is found.

The extractor program approximates a finite-state grammar developed to cover sentences in the test corpus while remaining as general as possible. A full context-free grammar is more desirable, but proved to be impractical in the time available to conduct this research. The program code for the extractor program is included as Appendix A.

The algorithm for the extractor is as follows. For a given natural-language sentence:

- Identify the first verb group that is not part of a clause or phrase. Verb groups with modals are preferred over those without.
- Segment the input into the front scope (the input left of the verb group), the verb group, and the back scope (the remainder of the input).
- Find the subject of the verb group in the front scope of the input. The subject and its modifiers should be in the last noun group that is not part of a phrase or clause.
- Find the verb, its modals, and its modifiers in the verb group.
- Find the object of the verb group in the back scope of the input. The object and its modifiers should be in the first noun group that is not part of a phrase or clause.
- Construct a meaning list listing the subject(s), the subject modifiers, the object(s) and modifiers, and the verb(s) and its modifiers including modals.

All three segments may contain modifiers, subclauses, and subphrases. The extractor does permit input that is not a complete sentence.

The input is first checked for a compound sentence; if so, its parts are processed separately. Appositives are then extracted. Modals or verbal groupings not part of a clause or phrase are sought as the main verb group. Bracketed noun groups are submitted to a recursive routine written by Professor Rowe and modified by the author, which identifies quantifiers, adjectives, and other attributes of the head noun (the subject of the group). The subparse routine also recognizes infinitive groupings as the subject (or object) if no bracketed noun groups were found. The verb group was parsed with a routine written by Professor Rowe that identifies modals, tense markers, adverbs, embedded objects, and conjunctions within the grouping.

Relevant subordinate terms are also identified; phrases and clauses are parsed using the split algorithm mentioned earlier. While it is relatively easy to identify the beginning of a phrase or clause, identifying the end is another matter. To address this problem, phrases and clauses are consecutively extracted from the rear of the scope fragment. We identify the last occurrence of a word that could begin a phrase or clause (preposition, pronoun, or adverb) and attribute the words following it as part of the phrase or clause.

D. SAMPLE OUTPUT FROM EXTRACTOR COMPONENT

This provides an example of the extractor operation:

- Input in natural-language format.

- **Information of questionable value to the general public must be evaluated before worldwide dissemination to assess the risk to the DoD.**

- LT CHUNK output indicated the part of speech of each input word as well as noun phrase and verb phrase groupings. (LT CHUNK uses with the Penn Treebank tag set [MARC1]. Appendix B provides a listing of the most significant tags used by LT CHUNK.)

  - **[Information_NN] of_IN [questionable_JJ value_NN] to_TO [the_DT general_JJ public_NN] < must_MD be_VB evaluated_VBN > before_IN [worldwide_JJ dissemination_NN] < to_TO assess_VB > [the_DT risk_NN] to_TO [the_DT DoD_NNP] ._.**

- The extractor program produced a meaning list that identified the main subject(s), object(s), and verbs of the input. Subordinate terms (from phrases and clauses) were also identified.

  - **[[main_subject_Group([subject(information), relationship(of), subj(value), modifier(value,questionable), relationship(to), subj(public), determiner(public,the), modifier(public,general)]), main_object_Group([relationship(before), obj(dissemination), modifier(dissemination,worldwide), obj(risk), obj(to_assess),**

determiner(risk,the), relationship(to), obj(dod),

determiner(dod,the)]),

main_verb_Group([modal(must), passive(must,be),

verb(must,evaluated)])]

Key terms from the meaning list can be identified for further processing. Modifiers are attributes of the subjects, objects, and verbs to which they refer. Any determiners provide for existential or universal quantification of the subject or object they modify. Subordinate terms and their relationships to their subject establish a hierarchical, *part-of*, *a-kind-of*, or a conditional relationship depending on the clause or phrase-head term.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. TESTING THE EXTRACTOR COMPONENT

This section summarizes the results of testing the extractor component. The following conditions were tabulated:

- Whether the meaning list (ML) correctly identifies main subject(s), verb(s), and objects(s) based on the natural-language input
- Whether the meaning list is incorrect but correctly identifies main constructs of input based on the tagger and intermediate output
- Whether the meaning list is incorrect

The test corpus was comprised of policy statements pertaining to web page content at the Naval Postgraduate School. Many of the policy statements were in a prescriptive format; they had a modal grouping as well as well-defined front and back scopes. However, quite a few statements were expository in nature; free-form with the intent to clarify a point. Some statements were bulleted items that were dependent clauses or phrases. Many of the statements contained technical constructs such as Uniform Resource Locator (URL) addresses.

Table 1 summarizes the results of the testing. Input sentences (excluding lists) contained twenty-two words on average; the average meaning list contained eighteen facts. Some words were combined and presented as unknown facts in a meaning list.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. CONCLUSIONS AND FUTURE WORK

A. CONCLUSIONS

In this thesis we developed an architecture for a natural-language processing tool to be used as a part of a policy workbench. We hypothesized that the NLIPT, when properly implemented, could minimize user interaction when formalizing policy statements. This is desirable when large policy bases are involved; it saves time and increases the likelihood that the system will be consistently used. As part of a partial proof of concept, we developed a prototype component, the extractor, for the NLIPT.

The extractor program proved adequate for parsing simple policy statements. It correctly identifies the subject, object, attributes, and the verb. However, the program must be refined if it is to adequately handle the complex sentences that exist in policy corpora.

The inadequacies of extractor program do not disprove our hypothesis that policy formalization can be largely automated. Nor do they invalidate the proposed architecture of the NLIPT. Our efforts must be directed to improving the extractor component and to implementing the other components of NLIPT. Modifying the program to use a full context-free grammar instead of ad hoc rules could increase the robustness of the program. The use of the partial tagger greatly simplified the algorithm of the extractor program; it also introduced modularity into the NLIPT that allows replacement or modifications without affecting the other components. Augmenting the tagger output by identifying phrases (prepositional, adverbial, etc.) before submitting it to the extractor

program could help. Also, examining the possible word-senses of the input based on the assigned tag could provide a means to correct mistagged words.

B. FUTURE WORK

Future research topics include the implementation and further refinement of the proposed components of the NLIPT. We incorporated a commercial-off-the-shelf (COTS) solution into the extractor. There may be suitable COTS solutions for all the components. Evaluation and testing of COTS solutions could significantly reduce the policy workbench development time.

Evaluation metrics for the NLIPT is another area of research. Some issues to address when developing metrics might include the following:

- How easy is it to reconstruct the policy from the schema?
- How accurate is the logical representation?
- How relevant are the indexed schema retrieved from the policy base?
- Should the schema be indexed?

The NLIPT is only one of many components of a larger system. Moreover, the NLIPT must be complemented with a natural-language response system, which remains to be investigated.

Another avenue of research is that of exploring the interaction or degree of coupling of the natural-language interface with the other policy-workbench tools. For example, thesis research is being conducted at the Naval Postgraduate School on the

automated testing of policy. For instance, the testing tools may place tool-specific requirements on the natural-language interface.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: EXTRACTOR CODE

```
/* File: Extractor.pl
   Author:  V.L.Ong
            Professor Rowe

   Written in B-Prolog

This program parses sentences to find subjects, objects, and verb
phrases
   INPUT:  Sentence - natural language input to be parsed
           Tags     - part of speech tags for Sentence
   OUTPUT: ML - Meaning List identifying the subject, object,
                main verb phrase

 */


preparse(Sentence,_,_) :-
      write('Preparsing the sentence: '), nl,
      write(Sentence), nl, fail.

/* process compound sentences */
preparse(Sentence, Tags, ML) :-
      is_compound_sentence(Sentence, Tags, S1, T1, S2, T2, Conj),
      preparse(S1, T1, ML1),
      preparse(S2, T2, ML2),
      Rel=..[relationship, Conj],
      append(ML1,[Rel],Temp),
      append(Temp, ML2, ML), !.


/* Segments opening clauses/phrases offset by comma from input
   Processes each separately, assumes first noun group following
   phrase is subject
*/

preparse(FS, FT, FML) :-
       append(T1, [','|T2], FT), length(T1, N1),
       (append(['IN'], D1, T1);
       append(['WRB'],D1,T1);append(['RB'],D1,T1)),
       append(S1,[','|S2], FS), length(S1,N1),
       append(PhrsHD, DS1, S1), length(PhrsHD,1),
       find_noun_group(T2,S2,NGT,NGS,_,_,_,_),
       extract_subject(NGT,NGS,Sub),
       interiorparse(DS1,D1,ML),
       (member(subject(S),Sub),CML = [relationship(S,PhrsHD)|ML];
        CML = [relationship(PhrsHD)|ML]),
       preparse(S2,T2,ML2),!,
       append(ML2,CML,FML),!.

/* Segments opening clause offset with comma
that starts with a determiner such as Which
*/
```

```
    preparse(FS, FT, FML) :-
       append(T1, [','|T2], FT), length(T1, N1),
       append(S1,[','|S2], FS), length(S1,N1),
       find_noun_group(T1,S1,NGT,NGS,_,_,_,_),
       NGT=['WDT'],
       find_noun_group(T2,S2,NGT2,NGS2,_,_,_,_),
       extract_subject(NGT2,NGS2,Sub),
       interiorparse(S1,T1,ML),!,
       (member(subject(S),Sub),CML = [relationship(S,NGS)|ML];
       CML = [relationship(NGS)|ML]),
       preparse(S2,T2,ML2),!,
       append(CML,ML2,FML),!.


/* Extracts appositives offset by commas from sentence
   and concatenates the remaining input for processing.
   Processes the appositive separately,
   Assumes noun group immediately preceeding appositive
   is subject of appositive

*/

preparse(Sentence, Tags, ML) :-
         append(T1, [','|T2], Tags), length(T1, N1),
       append(AposT,[','|T3],T2), length(AposT,N2),
       \+member(',',AposT),
       \+member('<',AposT),\+member('>',AposT),
       \+append(['CC'],DMY,T3),!,
       append(S1,[','|S2],Sentence), length(S1, N1), !,
       append(AposS,[','|S3],S2), length(AposS, N2), !,
       last_noun_group(T1,S1,LNGTags,LNGSent,_,_,_,_),
       extract_subject_np(LNGTags,LNGSent,Sub),
       interiorparse(AposS,AposT,ApML),
       (member(subject(S),Sub), append([apos_subj(S)],ApML,APML);
        append([apos],ApML,APML)),
       append(T1,T3,NTags),
       append(S1,S3,NSentence),!,
       preparse(NSentence, NTags, NML),
       append(NML,APML,ML),!.

/*segments a clause from a sentence and processes each
separately*/
preparse(Sentence, Tags, ML) :-
         append(T1, [','|T2], Tags), length(T1, N1),
       append(Clst,[','|T3],T2), length(Clst,N2),
       \+member(',',Clst), member('<',Clst),member('>',Clst),
       member('[',Clst),member(']',Clst),!,
       append(S1,[','|S2],Sentence), length(S1, N1),
       append(ClsS,[','|S3],S2), length(ClsS, N2), !,
         interiorparse(ClsS,Clst,ApML),
       append(T1,T3,NTags),
       append(S1,S3,NSentence),!,
       preparse(NSentence, NTags, NML),
       append(NML,ApML,ML),!.
```

```
/* Process sentence with modal*/
preparse(Sentence,Tags, ML) :-
        append(T1,['MD'|T2],Tags),
        splitscope(Sentence,Tags,'MD', SF, SVG, SB,TF,TVG ,TB),
      process_front_scope(SF,TF,Subject),
      process_back_scope(SB,TB, Object),
      process_modal_group(SVG, TVG, Modal, Verb),!,
      Subj=..[main_subject_Group,Subject],
      Obj=..[main_object_Group,Object],
      Vb =..[main_verb_Group,Verb],
      append([Subj],[Obj],Temp),
      append(Temp,[Vb],ML),!.


/* No modal but verb in sentence*/
preparse(Sentence,Tags,ML) :-
      member(Tagtype,Tags),verbtag(Tagtype),
      splitscope(Sentence,Tags,Tagtype,SF, SVG, SB,TF,TVG ,TB),
      process_front_scope(SF,TF,Subject),
      process_back_scope(SB,TB, Object),
      process_verb_group(SVG, TVG, Verb,VerbList),
      Subj=..[main_subject_Group,Subject],
      Obj=..[main_object_Group,Object],
      Vb=..[main_verb_Group,VerbList],
      append([Subj],[Obj],Temp),
      append(Temp,[Vb],ML),!.



/* no modal and no verb in sentence */
preparse(Sentence,Tags,ML) :- !,
        process_front_scope(Sentence,Tags,ML),!.


preparse(Sentence,Tags,[]) :- !.


/* SEGMENTING INPUT INTO FRONT SCOPE, BACK SCOPE, VERB GROUPING*/


/* Splitscope */
% Splits sentence into three parts based on verb grouping (verb
%or modal)
% Checks first to make sure verb grouping is not part of a clause

splitscope(Sentence, Tags, TagType, SFront,VerbGroup, SBack,
      TFront,VGTags, TBack):-
        not_part_of_clause(Sentence,Tags, TagType, FScopeLength),
        \+ infinitive(Sentence,Tags,TagType, Dmy),!,
        append(TFront,T1,Tags),
        length(TFront,FScopeLength),
        append(VGT, ['>'|TBack], T1),length(TBack, N2),
        append(VGT, ['>'], VGTags),
        append(SFront,L1,Sentence), length(SFront,FScopeLength),!,
```

```
        append(VerbGroup, SBack, L1), length(SBack,N2),!.



/*is_compound_sentence*/
      %This function succeeds if the submitted input
      %has the properties of a compound sentence.  Returns
      %the two independent clauses and  the conjunction
      %Input:  sentence, tags
      %Output: sentence1, tags1, sentence2, tags2, conjunction

is_compound_sentence(Sentence, Tags, S1, T1, S2, T2, Conj) :-
      (append(S1,[',',Conj|S2],Sentence);
       append(S1,[';',Conj|S2], Sentence)),
      (coorconj(Conj); conjadv(Conj)), length(S1, N1),
      member('[',S1),member(']',S1),member('[',S2),member(']',S2),
      member('<',S1),member('>',S1),member('<',S2),member('>',S2),
      append(T1,[',',CT|T2], Tags), length(T1, N1), !.

/* not_part_of_clause */
      % This function succeeds if the word immediately before
      % the grouping with the Tagtype is a not clause word.  Returns
      % length of list preceeding group with TagType

            % Handles verbs or modals
      not_part_of_clause(Fragment, Tags, TagType, FLength) :-
                  (verbtag(TagType); TagType = 'MD'),
                  append(Front,['<'|Back],Tags), length(Front,FLength),
                  append(CheckFront,[TagType|Back1],Back),
                  \+append(Dc1,['<'|Dc2],CheckFront),!,
                  append(SF,['<'|SB],Fragment),length(SF,FLength),
                  last(Front,MaybeCls),!,
                  \+cls(MaybeCls), \+tocls(MaybeCls),
                  last_noun_group(Front,SF,NGT1,NGS1,TF1,SF1,_,_),
                   (\+spcls(NGT1);spcls(NGT1),\+member('[',TF1)),!.

      %Handles with nouns, pronouns
      not_part_of_clause(Fragment, Tags, TagType, FLength) :-
                  nountag(TagType),!,
                  append(Front,['['|Back],Tags), length(Front,FLength),
                  append(CheckFront, [TagType|Back1],Back),
                  \+append(Dc1,['['|Dc2],CheckFront),
                  append(SF,['['|SB],Fragment),length(SF,FLength),
                  last(Front,MaybeCls),!,
                  \+cls(MaybeCls), \+tocls(MaybeCls),
                   last_noun_group(Front,SF,NGT1,NGS1,TF1,SF1,_,_),
                   (\+spcls(NGT1);spcls(NGT1),\+member('[',TF1)),!.

      % Checks to see if the verb grouping is actually an infinitive
      % Succeeds if it is an infinitive

      infinitive(Fragment, Tags, TagType, []) :-
            TagType = 'VB',!,
            append(Front,['<'|Back],Tags),
```

```
      append(VerbGp,['>'|Back1],Back),
      member('TO',VerbGp),!,member('VB',VerbGp).


/* FRONT SCOPE PROCESSING */


/* Process_front_scope
   Extracts the subject from the grouping, identifies any
   determiners, any adjectives (attributes) */


 process_front_scope([],[],[]).

 /*  clause as opening statement with 'IN' tag as first grouping
     Assumption: first noun group following comma is subject
 */
 process_front_scope(FS, FT, FML) :-
       append(T1, [','|T2], FT), length(T1, N1),
       append(['IN'], D1, T1),!,
       append(S1,[','|S2], FS), length(S1,N1),
       append(Clause, DS1, S1), length(Clause,1),
       interiorparse(DS1,D1,ML),
       find_noun_group(T2,S2,NounTags,NounGrp,_,_,_,_),
       extract_subject(NounTags, NounGrp, Subj),
       member(subject(S),Subj),
       CML = [relationship(S,Clause)|ML],
       process_front_scope(S2,T2,ML2),!,
       append(CML,ML2,FML),!.

 /*  clause as opening statement with 'WDT' tag as first grouping
       Form: brackets surround WDT
     Assumption: first noun group following comma is subject
 */
    process_front_scope(FS, FT, FML) :-
       append(T1, [','|T2], FT), length(T1, N1),
       append(['[','WDT',']'],D1,T1),!,
       append(S1,[','|S2], FS), length(S1,N1),!,
       append(Clause, DS1, S1), length(Clause,3),!,
       interiorparse(DS1,D1,ML),!,
       find_noun_group(T2,S2,NounTags,NounGrp,_,_,_,_),!,
       extract_subject(NounTags, NounGrp, Subj),!,
       member(subject(S),Subj),
       append(['['],[Cl|']'],Clause),
       CML = [relationship(S,Cl)|ML],
       process_front_scope(S2,T2,ML2),!,
       append(CML,ML2,FML),!.



  /* Segments rear phrase and sends it for subordinate processing
     Assumes noun group nearest phrase is subject of phrase.
     Assumes noun group that is nearest main verbal grouping that
         is not part of a clause is the subject of the verb.
      */
```

```
   /* identifies last occurance of clause word */
   process_front_scope(FS, FT, FML) :-
      get_last_cls(FT, Cls, BLength),
      append(T1, [Cls|T2], FT), (cls(Cls);tocls(Cls)),
      length(T2,BLength),!,
      append(S1, [SCls|S2],FS), length(S2,BLength),!,
      interiorparse(S2,T2,ML),!,
      PML = [relationship(SCls)|ML],
      process_front_scope(S1,T1,NML),!,
      append(NML,PML,FML),!.


   %This routine addresses multiple objects
 process_front_scope(FS, FT, FML) :-
      find_noun_group(FT,FS,NT,NS,FrontS,FrontT,RestT,RestS),
          extract_subject(NT,NS,Sub1),
          subs_headnoun_subject(Sub1,Sub2),
          first(RestT, X), (X=',';X='CC'),
          process_front_scope(RestS,RestT,ML1),
          interiorparse(FrontS,FrontT,NML),!,
          append(NML,Sub2,ML),
            append(ML,ML1,FML),!.

 process_front_scope(FS, FT, FML) :-
      find_noun_group(FT,FS,NT,NS,FrontS,FrontT,RestT,RestS),
      extract_subject(NT,NS,Sub2),!,
      subs_headnoun_subject(Sub2,Sub1),
      interiorparse(FrontS,FrontT,NML1),
      interiorparse(RestS,RestT,NML),!,
      append(NML1,Sub1,ML),
      append(ML,NML,FML),!.


/* only noun group supplied */
process_front_scope(FS, FT, FML) :-
       append(['['], Back, FT),
       last(Back,X), X= ']',!,
       delimit_list(FT,FS,'[',Ftags,Fsent),!,
       extract_subject_np(Ftags,Fsent,FML1),
       subs_headnoun_subject(FML1,FML),!.

/* only infinitive left */
process_front_scope(FS, FT, FML) :-
       first(FT,Y), Y= '<',
       last(FT,X), X= '>',
       member('TO', FT),
       member('VB', FT),
       delimit_list(FT,FS,'<',Ftags,Fsent),!,
       make_word_from_list(Fsent,Infinitive),
       FML = [subject(Infinitive)],!.
```

```
process_front_scope(FS,FT,FML) :-
      member(X,FT), punc(X),
      delete(X,FT,FT1), delete(X,FS,FS1),
      process_front_scope(FS1,FT1,FML),!.

process_front_scope(FS, FT, FML) :-
      length(FT,1), (FT=['VBG'];FT=['IN'];FT=['TO'];FT=['VBN']),
      FS=[X],
      FML=[relationship(X)],!.

process_front_scope(FS,FT, FML) :-
      length(FT,1), FT=['CC'],
      FS=[X],
      FML=[conj(X)],!.

process_front_scope(FS,FT,[]) :-
      length(FT,1), FT=[X],punc(X),!.

/*program identifies as unknown things it does not handle */
process_front_scope(FS, FT, FML) :-
       member(X,FT),\+punc(X),
       make_word_from_list(FS, Unknown),
       FML = [unknown(Unknown)],!.

process_front_scope(_,_,[]).


/* get_last_cls */
% finds last clause and returns position
% from the back up to (but not including) the cls tag


% makes sure infinitive with to is not selected
get_last_cls(Tags, Cls, BLength) :-
    reversa(Tags, RT),
    append(T1,[Cls|T2],RT), tocls(Cls),
    \+append(X,['VB'],T1),length(T1,BLength),!.

get_last_cls(Tags, Cls, BLength) :-
      reversa(Tags, RT),!,
      append(T1,[Cls|T2],RT), cls(Cls),length(T1, BLength),!.

 /* SUB PARSE  SECTION */

  /* Input: Tags - phrase tags to be processed
            Sent - phrase content
      NGT  - tags of noun group (no brackets) most likely subject
      NGS  - noun group (no brackets)
      OUTPUT: ML with subordinate phrase parsed

  */
 /* interiorparse */

 interiorparse([],[],[]).
```

```
/* clauses with modals*/
interiorparse(SentFrag, Tags, ML):-
           append(T1,['MD'|T2],Tags),
       splitinscope(SentFrag,Tags,'MD', SF, SVG, SB,TF,TVG ,TB),
            process_front_scope(SF,TF,Sub),
            member(subject(S),Sub),
            process_modal_group(SVG, TVG, Modal, Verb),
            delete(subject(S),Sub,Sub1),
            member(verb(V1,V2),Verb),
            append([subj(S)],Sub1,Sub2),
            sub_hd_subj(Sub2,V2,Subject),
            process_back_scope(SB,TB, Object),
           member(object(O1),Object),!,
           delete(object(O1),Object,OB),
           append([obj(O1)],OB,OB2),
           sub_hd_obj(OB2,V2,OB3),
           append(Subject,Verb,ML3),
           append(ML3, OB3, ML),!.

interiorparse(SentFrag, Tags, ML):-
           append(T1,['MD'|T2],Tags),
       splitinscope(SentFrag,Tags,'MD', SF, SVG, SB,TF,TVG ,TB),
            process_front_scope(SF,TF,Subject),
            member(subject(S),Subject),
            process_modal_group(SVG, TVG, Modal, Verb),
            member(verb(V1,V2),Verb),!,
            delete(subject(S),Subject,Sb),
            append([subj(S)],Sb,ML1),
            sub_hd_subj(ML1,V2,ML2),
            append(ML2,Verb, ML),!.

% no subject, but has back scope
interiorparse(SentFrag, Tags, ML) :-
      append(T1,['MD'|T2],Tags),
      splitinscope(SentFrag,Tags,'MD', SF, SVG, SB,TF,TVG ,TB),
      process_modal_group(SVG, TVG, Modal, Verb),
      member(verb(V1,V2),Verb),
      process_back_scope(SB,TB,Object),
      member(object(O1),Object),!,
      delete(object(O1),Object,OB),
      append([obj(O1)],OB,ML1),
      sub_hd_obj(ML1,V2,ML2),
      append(Verb,ML2,ML),!.


% only modal present, not front or back scope
 interiorparse(SentFrag,Tags, ML) :-
       member('MD', Tags),
       process_modal_group(SVG, TVG, Modal, ML),!.

/* clauses with verbs */
% subject, verb, object; splits on first verbform encountered
interiorparse(SentFrag,Tags,ML) :-
```

```
        append(T1,[Tagtype|T2],Tags),verbtag(Tagtype),
        \+infinitive(SentFrag,Tags,Tagtype,_),
       splitinscope(SentFrag,Tags,Tagtype, SF, SVG, SB,TF,TVG ,TB),
        process_front_scope(SF,TF,Subject),
        process_verb_group(SVG, TVG, Verb,VerbList),
        process_back_scope(SB,TB, Object),
        member(subject(S),Subject),
        member(verb(V1,V2),VerbList),
        delete(subject(S),Sujbect,S2),
        append([subj(S)],S2,S3),
        sub_hd_subj(S3,V2,SML),
        delete(object(O1),Object,OB1),
        append([obj(O1)],OB1,OB),
        sub_hd_obj(OB,V2,OML),
        append(SML,VerbList,Temp),!,
        append(Temp,OML, ML),!.

    % subject, verb, no object
     interiorparse(SentFrag, Tags, ML) :-
        append(T1,[Tagtype|T2],Tags),verbtag(Tagtype),
        \+infinitive(SentFrag,Tags,Tagtype,_),
       splitinscope(SentFrag,Tags,Tagtype, SF, SVG, SB,TF,TVG ,TB),
        process_front_scope(SF,TF,Subject),
        process_verb_group(SVG, TVG, Verb,VerbList),
        member(subject(S),Subject),
        member(verb(V1,V2),VerbList),!,
        delete(subject(S),Subject,S2),
        append([subj(S)],S2,S3),
        sub_hd_subj(S3,V2,SML),
        append(SML,VerbList, ML),!.

      % no subject, but has backscope
    interiorparse(SentFrag, Tags, ML) :-
          append(T1,[Tagtype|T2],Tags),verbtag(Tagtype),
         \+infinitive(SentFrag,Tags,Tagtype,_),
        splitinscope(SentFrag,Tags,Tagtype, SF, SVG, SB,TF,TVG ,TB),
          process_verb_group(SVG, TVG, Verb,VerbList),
          process_back_scope(SB,TB,Object),
          member(verb(V1,V2),VerbList),
          member(object(O1),Object),!,
          delete(object(O1),Object,OB),
          append([obj(O1)],OB,OML),
          sub_hd_obj(OML,V2,OML1),
          append(VerbList,OML1,ML),!.


     % only verb in phrase/clause
     interiorparse(SentFrag, Tags, ML) :-
          append(T1,[Tagtype|T2],Tags),verbtag(Tagtype),
          \+infinitive(SentFrag,Tags,Tagtype,_),
          process_verb_group(SVG, TVG, Verb,ML),!.


    /* clause with no verbs */
    interiorparse(SentFrag, Tags, ML) :-
```

```
        process_front_scope(SentFrag,Tags,Subject),
        member(subject(S),Subject),
        delete(subject(S),Subject,SB),
        append([subj(S)],SB,ML),!.

    %no brackets around phrase words
    interiorparse(SentFrag, Tags, ML) :-
        process_front_scope(SentFrag,Tags,Subject),
        ML=Subject,!.


    /* return nothing */
    interiorparse(_, _, []).

   /*interiorparseBS is the same as interiorparse, but handles the
     back scope */

     /* interiorparseBS */

     interiorparseBS([],[],[]).


     /* clauses with modals*/
     interiorparseBS(SentFrag, Tags, ML):-
          append(T1,['MD'|T2],Tags),
           splitinscope(SentFrag,Tags,'MD', SF, SVG, SB,TF,TVG ,TB),
                process_front_scope(SF,TF,Sub),
                member(subject(S),Sub),
                process_modal_group(SVG, TVG, Modal, Verb),
                delete(subject(S),Sub,Sub1),
                member(verb(V1,V2),Verb),
                append([subj(S)],Sub1,Sub2),
                sub_hd_subj(Sub2,V2,Subject),
                process_back_scope(SB,TB, Object),
               member(object(O1),Object),!,
               delete(object(O1),Object,OB),
               append([obj(O1)],OB,OB2),
               sub_hd_obj(OB2,V2,OB3),
               append(Subject,Verb,ML3),
               append(ML3, OB3, ML),!.

     interiorparseBS(SentFrag, Tags, ML):-
               append(T1,['MD'|T2],Tags),
           splitinscope(SentFrag,Tags,'MD', SF, SVG, SB,TF,TVG ,TB),
                process_front_scope(SF,TF,Subject),
                member(subject(S),Subject),
                process_modal_group(SVG, TVG, Modal, Verb),
                member(verb(V1,V2),Verb),!,
                delete(subject(S),Subject,Sb),
                append([subj(S)],Sb,ML1),
                sub_hd_subj(ML1,V2,ML2),
               append(ML2,Verb, ML),!.

     % no subject, but has backscope
     interiorparseBS(SentFrag, Tags, ML) :-
```

```
      append(T1,['MD'|T2],Tags),
      splitinscope(SentFrag,Tags,'MD', SF, SVG, SB,TF,TVG ,TB),
      process_modal_group(SVG, TVG, Modal, Verb),
      member(verb(V1,V2),Verb),
      process_back_scope(SB,TB,Object),
      member(object(O1),Object),!,
      delete(object(O1),Object,OB),
      append([obj(O1)],OB,ML1),
      sub_hd_obj(ML1,V2,ML2),
      append(Verb,ML2,ML),!.


% only modal
 interiorparseBS(SentFrag,Tags, ML) :-
       member('MD', Tags),
       process_modal_group(SVG, TVG, Modal, ML),!.

/* clauses with verbs */
% subject, verb, object; splits on first verbform encountered
interiorparseBS(SentFrag,Tags,ML) :-
    append(T1,[Tagtype|T2],Tags),verbtag(Tagtype),
    \+infinitive(SentFrag,Tags,Tagtype,_),
   splitinscope(SentFrag,Tags,Tagtype, SF, SVG, SB,TF,TVG ,TB),
     process_front_scope(SF,TF,Subject),
     process_verb_group(SVG, TVG, Verb,VerbList),
     process_back_scope(SB,TB, Object),
     member(subject(S),Subject),
     member(verb(V1,V2),VerbList),
     member(object(O1),Object),!,
     delete(subject(S),Sujbect,S2),
     append([subj(S)],S2,S3),
     sub_hd_subj(S3,V2,SML),
     delete(object(O1),Object,OB1),
     append([obj(O1)],OB1,OB),
     sub_hd_obj(OB,V2,OML),
     append(SML,VerbList,Temp),!,
     append(Temp,OML, ML),!.

% subject, verb, no object
 interiorparseBS(SentFrag, Tags, ML) :-
    append(T1,[Tagtype|T2],Tags),verbtag(Tagtype),
    \+infinitive(SentFrag,Tags,Tagtype,_),
   splitinscope(SentFrag,Tags,Tagtype, SF, SVG, SB,TF,TVG ,TB),
     process_front_scope(SF,TF,Subject),
     process_verb_group(SVG, TVG, Verb,VerbList),
     member(subject(S),Subject),
     member(verb(V1,V2),VerbList),!,
     delete(subject(S),Subject,S2),
     append([subj(S)],S2,S3),
     sub_hd_subj(S3,V2,SML),
     append(SML,VerbList, ML),!.

  % no subject, but has backscope
interiorparseBS(SentFrag, Tags, ML) :-
```

```
            append(T1,[Tagtype|T2],Tags),verbtag(Tagtype),
            \+infinitive(SentFrag,Tags,Tagtype,_),
           splitinscope(SentFrag,Tags,Tagtype, SF, SVG, SB,TF,TVG ,TB),
              process_verb_group(SVG, TVG, Verb,VerbList),
              process_back_scope(SB,TB,Object),
              member(verb(V1,V2),VerbList),
              member(object(O1),Object),!,
              delete(object(O1),Object,OB),
              append([obj(O1)],OB,OML),
              sub_hd_obj(OML,V2,OML1),
              append(VerbList,OML1,ML),!.

          % only verb in phrase/clause
           interiorparseBS(SentFrag, Tags, ML) :-
              append(T1,[Tagtype|T2],Tags),verbtag(Tagtype),
              \+infinitive(SentFrag,Tags,Tagtype,_),
               process_verb_group(SVG, TVG, Verb,ML),!.


          /* clause with no verbs */
          interiorparseBS(SentFrag, Tags, ML) :-
               process_back_scope(SentFrag,Tags,Object),
               member(object(O),Object),
               delete(object(O),Object,OB),
               append([obj(O)],OB,ML),
               !.

        %no brackets around phrase words
          interiorparseBS(SentFrag, Tags, ML) :-
               process_back_scope(SentFrag,Tags,Object),
               ML=Object,!.


          /* return nothing */
          interiorparseBS(_, _, []).


           /* Segments phrases based on first verb encountered
              Used for interior parsing of phrases*/

          /* splitinscope */
         % splits on first verb encountered

splitinscope(Sentence,Tags,TagType,SFront,VerbGroup,SBack,TFront,VGTag,
      TBack):-
               append(TFr,[TagType|Back],Tags),
               reversa(TFr,RFT),!,
               append(T1,['<'|TF1],RFT),!, length(TF1,N1),!,
               append(TFront,TF2,TFr), length(TFront,N1),!,
               append(TF2,[TagType],VGT),!,
               append(VGTB, ['>'|TBack], Back),length(TBack, N2),
               append(VGT, VGTB, VGTC),
               append(VGTC, ['>'], VGTags),!,
               append(SFront,L1,Sentence), length(SFront,N1),!,
               append(VerbGroup, SBack, L1), length(SBack,N2),!.
```

```
/* BACK SCOPE PROCESSING */


 /* process_back_scope */
% Finds the object of the verb

process_back_scope([],[],[]).


    /* identifies last occurance of clause word */
    process_back_scope(BS, BT, BML) :-
       get_last_cls(BT, Cls, BLength),
       append(T1, [Cls|T2], BT), (cls(Cls);tocls(Cls)),
       length(T2,BLength),!,
       append(S1, [SCls|S2],BS), length(S2,BLength),!,
       interiorparseBS(S2,T2,ML),!,
       PML = [relationship(SCls)|ML],
       process_back_scope(S1,T1,NML),!,
       append(NML,PML,BML),!.

%This routine addresses multiple objects
   process_back_scope(BS, BT, BML) :-
       find_noun_group(BT,BS,NT,NS,FrontS,FrontT,RestT,RestS),
       extract_object(NT,NS,Obj1),
       subs_headnoun_object(Obj1,OB1),
       first(RestT, X), (X=',';X='CC'),
       process_back_scope(RestS,RestT,ML1),
       interiorparseBS(FrontS,FrontT,NML),!,
       append(NML,OB1,ML),
       append(ML,ML1,BML),!.


    process_back_scope(BS, BT, BML) :-
       find_noun_group(BT,BS,NT,NS,FrontS,FrontT,RestT,RestS),
       extract_object(NT,NS,Sub1),
       subs_headnoun_object(Sub1,OB1),
       interiorparseBS(FrontS,FrontT,NML1),
       interiorparseBS(RestS,RestT,NML),!,
       append(NML1,OB1,ML),
       append(ML,NML,BML),!.


  /* only noun group supplied */
  process_back_scope(BS, BT, BML) :-
       append(['['], Back, BT),!,
       last(Back,X), X= ']',!,
       delimit_list(BT,BS,'[',Btags,BSent),!,
       extract_object_np(Btags,BSent,BML),!.

  /* only infinitive left */
  process_back_scope(BS, BT, BML) :-
       first(BT,Y), Y= '<',
```

```
                last(BT,X), X= '>',
                member('TO', BT),
                member('VB', BT),
                delimit_list(BT,BS,'<',Ftags,BSent),!,
                make_word_from_list(BSent,Infinitive),
                BML = [object(Infinitive)],!.


        %punctuation appears the same in both tag lists and sentences
        process_back_scope(FS, FT, FML) :-
               member(X,FT), punc(X),
               delete(X,FT,FT1), delete(X,FS,FS1),
               process_back_scope(FS1,FT1,FML), !.

         process_back_scope(BS, BT, BML) :-
                length(BT,1), (BT=['VBG'];BT=['TO'];BT=['IN']),
                BML =[relationship(BS)], !.

        process_back_scope(BS,BT,BML) :-
                length(BT,1), BT=['CC'],
                BML =[conj(BS)],!.

        process_back_scope(BS,BT,[]) :-
                length(BT,1), BT=[X],punc(X),!.

        process_back_scope(FS, FT, FML) :-
               member(X,FT),\+punc(X),
               make_word_from_list(FS, Unknown),
               FML = [unknown(Unknown)],!.

        process_back_scope(_,_,[]) :- !.



        /* NOUN GROUP PROCESSING */


        /*find_noun_group
          Returns first noun group in fragment
          Returned with brackets*/

        % Finds first noun group in fragment.
        % Noun group returned with brackets
      find_noun_group(TagsFrag, SentFrag, NGTags, NGSent,SFront,TFront,
TRest, SRest):-
             append(TFront,['['|TB],TagsFrag),
             length(TFront,N1), !,
             append(TFront,T1,TagsFrag), !,
             append(NGT, [']'|TRest], T1),length(TRest, N2),!,
             append(NGT, [']'], NGTags),
             append(SFront,L1,SentFrag), length(SFront,N1),!,
             append(NGSent, SRest, L1), length(SRest,N2),!.
```

```
%Finds last noun group in fragment and extracts it from the submitted
%fragment
       %Noun group returned without brackets

       last_noun_group([],[],[],[],[],[],[],[]).

       %returns infinitive if noun group is not behind it
      last_noun_group(Tags, Sent, LNGTags, LNGSent, TFront,
            SFront, TRest, SRest):-
             reversa(Tags,RTags),
            append(RTback,['>'|Tb],RTags),
            \+member('[',RTback),\+member(']',RTback),
             length(RTback,N1),
            append(RLNGTags,['<'|RTfront],Tb),
            member('TO',RLNGTags),member('VB',RLNGTags),
             reversa(Sent,RSent),
             append(RSback,['>'|Sb],RSent), length(RSback,N1),
             append(RLNGSent,['<'|RSfront],Sb),
             reversa(RLNGTags,LNGTags),
             reversa(RTfront,TFront),
             reversa(RTback,TRest),
             reversa(RLNGSent,LNGSent),
             reversa(RSback,SRest),
             reversa(RSfront,SFront),!.

       %returns the last noun group in the fragment submitted
last_noun_group(Tags, Sent, LNGTags, LNGSent, TFront, SFront, TRest,
          SRest):-
             reversa(Tags,RTags),
             append(RTback,[']'|Tb],RTags),
             append(RLNGTags,['['|RTfront],Tb),
             reversa(Sent,RSent),
             append(RSback,[']'|Sb],RSent),
             append(RLNGSent,['['|RSfront],Sb),
             reversa(RLNGTags,LNGTags),
             reversa(RTfront,TFront),
             reversa(RTback,TRest),
             reversa(RLNGSent,LNGSent),
             reversa(RSback,SRest),
             reversa(RSfront,SFront),!.

        last_noun_group(_,_,[],[],[],[],[],[]).



       /* extract_subject */
       % Extracts subject from noun group
        extract_subject(NT, NS, Subj) :-
          delimit_list(NT, NS, '[',NTags, NSent),
          extract_subject_np(NTags, NSent, Subj).

       /* extract_object */
        % Extracts object from noun group
         extract_object(NT, NS, Obj) :-
           delimit_list(NT, NS, '[',NTags, NSent),
```

```
        extract_object_np(NTags, NSent, Obj).

      /* delimit_list */
       % Extracts tags, and words from single grouping

       delimit_list(TagList, SentList, Delim, TagsOnly, SentOnly) :-
            butlast(TagList, TL1), butlast(SentList, SL1),
            append(_,[Delim|TagsOnly],TL1),
           \+ member(Delim, TagsOnly),
            append(_,[Delim|SentOnly],SL1),
            \+ member(Delim, SentOnly),!.


    /*  Extracts subject from unbracketed noun group,
        Identifies modifiers and determiners
        Written by Professor Rowe and modified by author
    */

    /* extract_subject_np */
extract_subject_np([T],[S],[subject(S)]) :- nountag(T), !.
extract_subject_np([T],[S],[subject(S)]) :- T='VBG',!.
extract_subject_np([T],[S],[anaphoric(S),subject(S)]) :- T='WDT',!.
extract_subject_np([T],[S],[subject(S)]) :- T='CD',!.
extract_subject_np([T],[S],[anaphoric(S),subject(S)]) :- spPrn(S),!.
extract_subject_np([T1|TL],[S1|SL],[determiner(noun,S1)|ML]) :-
           adjtag(T1), T1='DT',!, extract_subject_np(TL,SL,ML), !.
extract_subject_np([T1|TL],[S1|SL],[modifier(noun,S1)|ML]) :-
           adjtag(T1), !, extract_subject_np(TL,SL,ML), !.

extract_subject_np([T1,T2|TL],[S1,S2|SL],[modifier(noun,S1)|ML]) :-
          nountag(T1), nountag(T2), !,
extract_subject_np([T2|TL],[S2|SL],ML), !.
extract_subject_np([T1,T2|TL],[S1,S2|SL],[modifier(S2,S1)|ML]) :-
           T1='RB', extract_subject_np([T2|TL],[S2|SL],ML), !.

extract_subject_np([T1,T2|TL],[S1,S2|SL],[relationship(S1,S2)]) :-
           (nountag(T1); T1='CD'), T2='IN',!,
           extract_subject_np(TL,SL,ML), !.
extract_subject_np([T1|TL],[S1|SL],ML):-
           extract_subject_np(TL,SL,ML), !.



/* extract_object_np */
extract_object_np([T1|TL],[S1|SL],[determiner(noun,S1)|ML]) :-
           adjtag(T1), T1='DT',!, extract_object_np(TL,SL,ML), !.
extract_object_np([T1|TL],[S1|SL],[modifier(noun,S1)|ML]) :-
           adjtag(T1), !, extract_object_np(TL,SL,ML), !.

extract_object_np([T1,T2|TL],[S1,S2|SL],[modifier(noun,S1)|ML]) :-
            nountag(T1), nountag(T2), !,
extract_object_np([T2|TL],[S2|SL],ML), !.
extract_object_np([T],[S],[object(S)]) :- nountag(T), !.
extract_object_np([T],[S],[object(S)]) :- T='VBG',!.
```

```
extract_object_np([T],[S],[object(S)]) :- T='WDT',!.
extract_object_np([T],[S],[object(S)]) :- spPrn(S),!.
extract_object_np([T1,T2|TL],[S1,S2|SL],[modifier(S2,S1)|ML]) :-
            T1='RB', extract_object_np([T2|TL],[S2|SL],ML), !.
extract_object_np([T1,T2|TL],[S1,S2|SL],[object(verb,S1)]) :-
            nountag(T1), (T2='IN'; T2='VBG'; T2='VBN'),
            !,extract_object_np(TL,SL,ML), !.
extract_object_np([T1|TL],[S1|SL],ML) :- extract_object_np(TL,SL,ML),
            !.


/* process_modal_group */
% This routine assumes only a modal grouping in SVG, TVG
process_modal_group(SVG, TVG, Modal, Verb) :-
          delimit_list(TVG, SVG, '<', ModalTags, ModalSent),!,
          append(T1,['MD'|T2],TVG), length(T1,N1),!,
          append(S1,[Modal|S2],SVG), length(S1,N1),!,
          polpreparse_modverb(ModalSent, ModalTags, Modal, List,_,_),
          Verb = [modal(Modal)|List] ,!.


/* process_verb_group*/
 % This routine assumes only a verb grouping in SVG, TVG
process_verb_group(SVG, TVG, Verb, ML) :-
           delimit_list(TVG, SVG, '<', VerbTags, VerbSent),!,
           append(T1,[X|T2],TVG), verbtag(X), length(T1,N1),!,
           append(S1,[Verb|S2],SVG), length(S1,N1),!,
          parse_verb(VerbSent, VerbTags, Verb, ML),!.



/* This routine identifies the verb, modal and any modifiers in
        a verb group.

         WRITTEN:  PROFESSOR ROWE
*/
polpreparse_modverb([Adverb|S],['RB'|Tags],Modal,
      [modifier(Modal,Adverb)|ML],SB,TB) :-
                !,polpreparse_modverb(S,Tags,Modal,ML,SB,TB).
polpreparse_modverb([Verb,Adverb|S],[Verbtag,'RB'|Tags],Modal,
      [modifier(Modal,Adverb)|ML],SB,TB) :-
            verbtag(Verbtag), \+(first(Tags,Nexttag),
            verbtag(Nexttag)),!,
            polpreparse_modverb([Verb|S],[Verbtag|Tags],Modal,ML,SB,TB.
polpreparse_modverb([Noun|S],[Nountag|Tags],Modal,[subject(Modal,Noun)|
      ML],SB,TB):-
      nountag(Nountag), !, polpreparse_modverb(S,Tags,Modal,ML,SB,TB).
polpreparse_modverb([Beform|S],[Verbtag|Tags],Modal,
      [passive(Modal,Beform)|ML],SB,TB):-
            verbtag(Verbtag), beform(Beform), \+first(Tags,'CC'),
            member(Verbtag2,Tags), verbtag(Verbtag2), !,
            polpreparse_modverb(S,Tags,Modal,ML,SB,TB).
polpreparse_modverb([Haveform|S],[Verbtag|Tags],Modal,
      [passive(Modal,Haveform)|ML],SB,TB):-
            verbtag(Verbtag), haveform(Beform), \+ first(Tags,'CC'),
```

```
            member(Verbtag2,Tags), verbtag(Verbtag2), !,
            polpreparse_modverb(S,Tags,Modal,ML,SB,TB).
polpreparse_modverb([Verb1,Conj,Verb2|S],[Verbtag,'CC',Verbtag|Tags],
      Modal,[verb(Modal,Verb12)],S,Tags) :-
            verbtag(Verbtag), name(Verb1,AV1), name(Verb2,AV2),
            name('_and_',AV0),append(AV1,AV0,AV10),
            append(AV10,AV2,AV12), name(Verb12,AV12), !.
polpreparse_modverb([Verb|S],[Verbtag|Tags],Modal,[verb(Modal,Verb)],S,
      Tags) :-
         verbtag(Verbtag), !.
polpreparse_modverb([_|S],[PS|Tags],Modal,ML,SB,TB) :-
         polpreparse_modverb(S,Tags,Modal,ML,SB,TB).


/*Same routine as above but no modal in verb group*/

parse_verb([Adverb|S],['RB'|Tags],Verb,[modifier(Verb,Adverb)|ML]):-  !,
         parse_verb(S,Tags,Verb,ML).
parse_verb([Vb,Adverb|S],[Verbtag,'RB'|Tags],Verb,
            [modifier(Verb,Adverb)|ML]) :-
                  verbtag(Verbtag), \+(first(Tags,Nexttag),
                  verbtag(Nexttag)),!,
                  parse_verb([Vb|S],[Verbtag|Tags],Verb,ML).
parse_verb([Noun|S],[Nountag|Tags],Verb,
            [subject(Verb,Noun)|ML]):-
                  nountag(Nountag), !, parse_verb(S,Tags,Verb,ML).
parse_verb([Beform|S],[Verbtag|Tags],Verb,
            [passive(Verb,Beform)|ML]):-
                  verbtag(Verbtag), beform(Beform), \+first(Tags,'CC'),
            member(Verbtag2,Tags), verbtag(Verbtag2), !,
            parse_verb(S,Tags,Verb,ML).
parse_verb([Haveform|S],[Verbtag|Tags],Verb,
      [passive(Verb,Haveform)|ML]):-
            verbtag(Verbtag), haveform(Beform), \+ first(Tags,'CC'),
            member(Verbtag2,Tags), verbtag(Verbtag2), !,
            parse_verb(S,Tags,Verb,ML).
parse_verb([Verb1,Conj,Verb2|S],[Verbtag,'CC',Verbtag|Tags],Verb,
            [verb(Verb12,Verb12)]) :-
            verbtag(Verbtag), name(Verb1,AV1), name(Verb2,AV2),
            name('_and_',AV0),append(AV1,AV0,AV10),
            append(AV10,AV2,AV12), name(Verb12,AV12), !.
parse_verb([Vb|S],[Verbtag|Tags],Verb,[verb(Verb,Vb)]) :-
            verbtag(Verbtag), !.
parse_verb([_|S],[PS|Tags],Verb,ML) :-
            parse_verb(S,Tags,Verb,ML).


%FACTS

      verbtag('VB').
      verbtag('VBD').
      verbtag('VBG').
      verbtag('VBN').
      verbtag('VBP').
      verbtag('VBZ').
```

```
nountag('NN').
nountag('NNS').
nountag('NNP').
nountag('NNP').
nountag('NNPS').
nountag('PRP').
nountag('FW').
nountag('WP').

adjtag('JJ').
adjtag('JJR').
adjtag('JJS').
adjtag('CD').
adjtag('DT').
adjtag('PDT').
adjtag('POS').
adjtag('PRP$').
adjtag('WP$').
adjtag('RB').
adjtag('VBG').

spcls('WDT').
tocls('TO').
cls('IN').
cls('WP').
cls('WRB').
cls('WP$').

beform('is').
beform('are').
beform('was').
beform('were').
beform('be').
beform('being').
beform('been').
haveform('has').
haveform('have').
haveform('had').
haveform('having').

clauseword('who').
clauseword('that').
clauseword('which').
clauseword('whom').
clauseword('whose').

coorconj('and').
coorconj('but').
coorconj('nor').
coorconj('or').
coorconj('for').
coorconj('yet').

conjadv('also').
```

```
conjadv('besides').
conjadv('consequently').
conjadv('furthermore').
conjadv('however').
conjadv('moreover').
conjadv('nevertheless').
conjadv('otherwise').
conjadv('then').
conjadv('therefore').
conjadv('thus').
conjadv('still').

%May be used as adj or pronouns, as well as other
%parts of speech
spPrn(all).
spPrn(another).
spPrn(any).
spPrn(both).
spPrn(each).
spPrn(either).
spPrn(few).
spPrn(many).
spPrn(more).
spPrn(neither).
spPrn(one).
spPrn(other).
spPrn(several).
spPrn(some).
spPrn(that).
spPrn(these).
spPrn(this).
spPrn(those).
spPrn(what).
spPrn(which).

punc(':').
punc('.').
punc(';').
punc('(').
punc(')').
punc('[').
punc(']').
punc('<').
punc('>').
punc(').').
punc(',').
punc('"').


/* Utility functions
  Written by: Professor Rowe*/

%Combines list contents into one item with underscores between
%the original items
make_word_from_list([S],S):- !.
```

```
      make_word_from_list([S1|SL],S) :-
            make_word_from_list(SL,S2),
            name(S2,AS2), name(S1, AS1),
            append(AS1,[95|AS2], AS12),
            name(S, AS12),!.

      butlast(L,NL) :- append(NL,[_],L), !.

      first([X|_],X).

      last([X],X) :- !.
      last([_|L],X) :- last(L,X).

      delete(X,[],[]) :- !.
      delete(X,[X|L],NL) :- !, delete(X,L,NL), !.
      delete(X,[Y|L],[Y|NL]) :- delete(X,L,NL), !.

      reversa(L,RL) :- reversa2(L,[],RL), !.
      reversa2([],L,L).
      reversa2([X|L],L2,RL) :- reversa2(L,[X|L2],RL).

%Written by Professor Rowe and modified by Ong
      % The following routines are used to clean up lists from
      %extract_subject_np, extract_object_np, and the equivalent
      %routines for the phrases.  They relate the modifiers to the
      %modified noun, and places the head noun at the list head

substitute_headnoun_subject(ML,Verb,NML) :-
            member(subject(Subject),ML),
             substitute_headnoun2(ML,Subject,ML3), NML =
            [subject(Verb,Subject)|ML3], !.
substitute_headnoun_subject(_,_,[]):-!.

substitute_headnoun2([],_,[]).
substitute_headnoun2([modifier(noun,M)|ML],Subject,[modifier(Subject,M)
      |NML]):-
            !, substitute_headnoun2(ML,Subject,NML).
substitute_headnoun2([determiner(noun,M)|ML],Subject,
      [determiner(Subject,M)|NML]):- !,
            substitute_headnoun2(ML,Subject,NML).
substitute_headnoun2([M1|ML],Subject,[M1|NML]):-
            !, substitute_headnoun2(ML,Subject,NML).


sub_hd_subj(ML,Verb,NML) :-
         member(subj(Subject),ML),
         sub_hd_subj2(ML,Subject,ML3), NML = [subj(Verb,Subject)|ML3],
            !.
sub_hd_subj(_,_,[]):-!.
sub_hd_subj2([],_,[]).
sub_hd_subj2([modifier(noun,M)|ML],Subject,[modifier(Subject,M)|NML]):-
            !, sub_hd_subj2(ML,Subject,NML).
sub_hd_subj2([determiner(noun,M)|ML],Subject,
      [determiner(Subject,M)|NML]):-
```

```
      !, sub_hd_subj2(ML,Subject,NML).
sub_hd_subj2([M1|ML],Subject,[M1|NML]):-
      !, sub_hd_subj2(ML,Subject,NML).



subs_headnoun_subject(ML1,NML) :-
         member(subject(Subject),ML1),delete(subject(Subject),ML1,ML),
        subs_headnoun2(ML,Subject,ML3), NML = [subject(Subject)|ML3],
        !.
subs_headnoun_subject(_,_,[]):-!.
subs_headnoun2([],_,[]).
subs_headnoun2([modifier(noun,M)|ML],Subject,[modifier(Subject,M)|NML]
     :-
        !, subs_headnoun2(ML,Subject,NML).
subs_headnoun2([determiner(noun,M)|ML],Subject,
      [determiner(Subject,M)|NML]):-
            !, subs_headnoun2(ML,Subject,NML).
subs_headnoun2([M1|ML],Subject,[M1|NML]):-
            !, subs_headnoun2(ML,Subject,NML).



subs_headnoun_object(ML1,NML) :-
         member(object(Object),ML1),delete(object(Object),ML1,ML),
         subs_headnoun2(ML,Object,ML3), NML = [object(Object)|ML3], !.
subs_headnoun_object(_,_,[]):-!.

subs_headnoun2([],_,[]).
subs_headnoun2([modifier(noun,M)|ML],Object,
      [modifier(Object,M)|NML]):-
            !, subs_headnoun2(ML,Object,NML).
subs_headnoun2([determiner(noun,M)|ML],Object,
      [determiner(Object,M)|NML]):-
            !, subs_headnoun2(ML,Object,NML).
subs_headnoun2([M1|ML],Object,[M1|NML]):- !,
            subs_headnoun2(ML,Object,NML).


substitute_headnoun_object(ML,Verb,NML) :-
         member(object(Object),ML),
         sub_obj2(ML,Object,ML3), NML = [object(Verb,Object)|ML3], !.
substitute_headnoun_object(_,_,[]):- !.

sub_obj2([],_,[]).
sub_obj2([modifier(noun,M)|ML],Object,[modifier(Object,M)|NML]):-
                  !, sub_obj2(ML,Object,NML).
sub_obj2([determiner(noun,M)|ML],Object,
      [determiner(Object,M)|NML]):-
            !, sub_obj2(ML,Object,NML).
sub_obj2([M1|ML],Object,[M1|NML]):-
      !, sub_obj2(ML,Object,NML).


sub_hd_obj(ML,Verb,NML) :-
```

```
        member(obj(Object),ML),
        sub2(ML,Object,ML3), NML = [obj(Verb,Object)|ML3], !.
sub_hd_obj(_,_,[]) :- !.
      sub2([],_,[]).
      sub2([modifier(noun,M)|ML],Object,[modifier(Object,M)|NML])
           :- !, sub2(ML,Object,NML).
      sub2([determiner(noun,M)|ML],Object,[determiner(Object,M)|NML])
           :- !, sub2(ML,Object,NML).
      sub2([M1|ML],Object,[M1|NML])
           :- !, sub2(ML,Object,NML).
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B: PENN TREEBANK TAG-SET (CONDENSED)

FROM: [LTG3]

POS Tag	Description	Example
CC	coordinating conjunction	and
CD	cardinal number	1, third
DT	determiner	the
EX	existential there	*there* is
FW	foreign word	d'hoevre
IN	preposition/subordinating conjunction	in, of, like
JJ	adjective	green
JJR	adjective, comparative	greener
JJS	adjective, superlative	greenest
LS	list marker	1)
MD	modal	could, will
NN	noun, singular or mass	table
NNS	noun plural	tables
NNP	proper noun, singular	John
NNPS	proper noun, plural	Vikings
PDT	predeterminer	*both* the boys
POS	possessive ending	friend*'s*
PRP	personal pronoun	I, he, it
PRP$	possessive pronoun	my, his
RB	adverb	however, usually, naturally, here, good
RBR	adverb, comparative	better
RBS	adverb, superlative	best
RP	particle	give *up*
TO	to	*to* go, *to* him
UH	interjection	uhhuhhuhh
VB	verb, base form	take
VBD	verb, past tense	took
VBG	verb, gerund/present participle	taking
VBN	verb, past participle	taken

POS Tag	Description	Example
VBP	verb, sing. present, non-3d	take
VBZ	verb, 3rd person sing. present	takes
WDT	wh-determiner	which
WP	wh-pronoun	who, what
WP$	possessive wh-pronoun	whose
WRB	wh-abverb	where, when

APPENDIX C: SAMPLE OUTPUT FROM THE EXTRACTOR

This section provides two more examples of correct output obtained from the prototype extractor component that most accurately reflects the structure of the input text.

1. Sentence with multiple subjects and objects and modal group.

    a. INPUT

Analysis and recommendations concerning lessons learned which would reveal sensitive military operations, exercises or vulnerabilities.

    b. TAGGER OUTPUT

[Analysis_NNP] and_CC [recommendations_NNS] concerning_VBG
[lessons_NNS] < learned_VBD > [which_WDT]
< would_MD reveal_VB > [sensitive_JJ military_JJ operations_NNS] ,_,
[exercises_NNS] or_CC [vulnerabilities_NNS] ._.

    c. MEANING LIST

[main_subject_Group([subject(analysis), conj(and), subject(recommendations),
subj(learned,lessons), subj(lessons), verb(learned,learned), obj(learned,which),
obj(which)]),
main_object_Group([object(operations), modifier(operations,sensitive),
modifier(operations,military), object(exercises), conj([or]),
object(vulnerabilities)]),
main_verb_Group([modal(would),verb(would,reveal)])]

    d. COMMENTS

The extractor successfully identified each subject of the main verb as well as each object and the appropriate modifier. The facts with predicates "subj" and "obj" are part of a phrase or clause. The participial term "concerning" is omitted however, but the main terms are identified.

2. Sentence with modal and phrases.

    a. INPUT

Information of questionable value to the general public must be evaluated before worldwide dissemination to assess the risk to the DoD.

    b. TAGGER OUTPUT

[Information_NN] of_IN [questionable_JJ value_NN] to_TO [the_DT
general_JJ public_NN] < must_MD be_VB evaluated_VBN > before_IN
[worldwide_JJ dissemination_NN] < to_TO assess_VB >
[the_DT risk_NN] to_TO [the_DT DoD_NNP] ._.

c. OUTPUT

[main_subject_Group([subject(information),relationship(of),subj(value),modifier(value,questionable),relationship(to),subj(public),determiner(public,the),modifier(public,general)]),
main_object_Group([relationship(before),obj(dissemination),modifier(dissemination,worldwide),obj(risk),obj(to_assess),determiner(risk,the),relationship(to),obj(dod),determiner(dod,the)]),
main_verb_Group([modal(must),passive(must,be),verb(must,evaluated)])]

d. COMMENTS

The extractor correctly identified the main subject of the statement, the main verb grouping, and the objects that were part of a phrase. Notice that the infinitive "to assess" was also identified. Phrase heads were presented as relationship predicate facts. Notice that the main subject is at the head of the list.

LIST OF REFERENCES

[ALLE95] Allen, James, *Natural Language Understanding*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1995.

[BARK98] Barker, K., S. Delisle, and S. Szpakowicz, "Test-Driving TANKA: Evaluating a Semi-Automatic System of Text Analysis for Knowledge Acquisition." *Proceedings of the 12th Canadian Artificial Intelligence Conference-CAI-98,* Vancouver, BC (Canada), June 18-20 1998, pp. 60-71.

[CHOV97] Cholvy, Laurence and Frederic Cuppens, "Analyzing Consistency of Security Policies," in *Proceedings of the IEEE Symposium on Security and Privacy*, 1997, pp.103-112.

[CHUN95] Chung, Minhwa and Dan I. Moldovan, "Parallel Natural Language Processing on a Semantic Network Array Processor," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No. 3, June 1995, pp. 391-406.

[DAMI01] Damianou, N., N. Dulay, E. Lupu, and M. Sloman, "The Ponder Specification Language," in Lecture Notes in Computer Science No. 1995, Springer-Verlag, Berlin, 2001, pp. 18-38.

[DELA93] Delannoy, J. F., C. Feng, S. Matwin, and S. Szpakowicz, "Knowledge Extraction from Text: Machine Learning for Text-to-rule Translation." *Proceedings of the Machine Learning Text Analysis Workshop,* European Conference on Machine Learning (ECML-93), pp. 1-7.

[DELA94] Delannoy, Jean-François and Riverson Rios, "Translating a detailed linguistic semantic representation into Horn clause logic." Brazilian Symposium on Artificial Intelligence (SBIA), Fortaleza, Ceara, Brazil, October 1994.

[GROV1] Grover, Claire, Colin Matheson, and Andrei Mikheev, "TTT: Text Tokenisation Tool." Language Technology Group, 2 Buccleuch Place, Edinburgh EH8 9LW, UK
http://www.ltg.ed.ac.uk/software/ttt/tttdoc.html#CHAPLTPOS

[HOPC79] Hopcraft, John E. and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley Publishing Company, Inc., Menlo Park, California, 1979.

[LTG1] Language Technology Group, "LTG software: LT CHUNK." Language Technology Group, 2 Buccleuch Place, Edinburgh EH8 9LW, UK http://www.ltg.ed.ac.uk/software/chunk/index.html

[LTG2] Language Technology Group, "LTG software: LT POS." Language Technology Group, 2 Buccleuch Place, Edinburgh EH8 9LW, UK. http://www.ltg.ed.ac.uk/software/pos

[LTG3] Language Technology Group, "LTG software: Penn Treebank Tag-Set." Language Technology Group, 2 Buccleuch Place, Edinburgh EH8 9LW, UK. http://www.ltg.ed.ac.uk/software/penn.html

[MARC1] Marcus, M., Beatrice Santorini, and M.A. Marcinkiewicz, "Building a large annotated corpus of English: The Penn Treebank," in *Computational Linguistics*, Vol. 19, No. 2, pp313-330.

[MICH91] Michael, James Bret, Edgar H. Sibley, and Richard L. Wexelblat, "A Modeling Paradigm for Representing Intentions in Information Systems Policy." *Proceedings of the First Workshop of Information Technologies and Systems*, Massachusetts Institute of Technology Sloan School of Management, Cambridge, Massachusetts. 1991, pp. 21-34.

[MICH93] Michael, James Bret, Edgar H. Sibley, Richard F. Baum, and Fu Li, "On the Axiomatization of Security Policy: Some Tentative Observations About Logic Representation," in *Database Security, VI: Status and Prospects*, North-Holland, Amsterdam, 1993, pp. 367-386.

[MICH93A] Michael, James Bret, "A Formal Process for Testing the Consistency of Composed Security Policies," Ph.D. dissertation, George Mason University, Fairfax, Virginia, 1993.

[MOFF91] Moffett, Jonathan D. and Morris. S. Sloman, "The Representation of Policies as System Objects." Domino Report: B1/IC/6.1, 20 Aug 91, in *Proceedings of the Conference on Organizational Computer Systems*, SIGIOS Bulletin Vol. 12, Nos. 2 & 3, pp. 171-184.

[MOUL92] Moulin, Bernard and Daniel Rousseau, "Automated Knowledge Acquisition from Regulatory Texts," in *IEEE,* Vol. 7, No. 5, October 1992, pp. 27-35.

[MOUL94] Moulin, Bernard and Daniel Rousseau, "SACD: A System for Acquiring Knowledge From Regulatory Texts," *Computers Elect. Engng*, Vol. 20, No. 2, 1994, pp. 131-149.

[RAND93] Flexner, Struart Berg Ed., *Random House Unabridged Dictionary*, Second Edition, Random House, Incorporated, New York, 1993.

[ROWE1] Rowe, Neil C., "Understanding of Technical Captions via Statistical Parsing," http://www.cs.nps.navy.mil/research/marie/nlang.html

[RUSS95] Russell, Stuart J. and Peter Norvig, *Artificial Intelligence A Modern Approach*, Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[SERG86] Sergot, M.J., F. Sadri, R. A. Kowalski, F. Kriwaczek, P. Hammond, and H. T. Cory. "The British Nationality Act as a Logic Program," *Communications of the ACM*, Vol. 29, No. 5, May 1986, pp. 370- 386.

[SIBL92] Sibley, Edgar H., James Bret Michael, and Richard L. Wexelblat, "Use of an Experimental Policy Workbench: Description and Preliminary Results," in *Database Security, V: Status and Prospects*, Elsevier Science Publishers (North-Holland), Amsterdam, 1992, pp.47-76.

[SLOA91] Sloane, S. B., "The Use of Artificial Intelligence by the United States Navy: Case Study of a Failure." *AI Magazine*, Vol. 12, No. 1, 1991, pp.80-92.

[STAN1] *Stanford Encyclopedia of Philosophy*, http://plato.stanford.edu/entries/logic-modal

[STON00] Stone, G. N., "A Path-based Network Policy Language," Ph. D. dissertation, Naval Postgraduate School, 2000.

[STRA99] Strassner, J., E. Ellensson, and B. Moore (editor), Policy Framework Core Information Model, Internet Engineering Task Force: Network Working Group, Internet Draft draft-ietf-policy-core-schema-02.txt, February 1999.

[TAN00] Tan, Yao-Hua and Walter Thoen, "INCAS: a legal expert system for contract terms in electronic commerce." *Decision Support Systems 29,* 2000, pp. 389-411.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Road, Suite 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Road
   Monterey, CA 93943-5101

3. Dean Dan Boger 1
   Chair, C3 Academic Group
   Naval Postgraduate School
   Monterey, CA 93943

4. Professor J. Bret Michael 2
   Department of Computer Science,
   Code CS/Mj
   Naval Postgraduate School
   Monterey, CA 93943-5118

5. Professor Neil C. Rowe 2
   Department of Computer Science,
   Code CS/Rn
   Naval Postgraduate School
   Monterey, CA 93943-5118

6. LT Vanessa Ong 2
   Commander, Naval Surface Reserve Force (N6)
   4400 Dauphine Street
   New Orleans, LA 70146-5100

7. Mr. Terry Mayfield 1
   Computer and Software Engineering Division
   Instituted for Defense Analysis
   1801 North Beauregard Street
   Alexandria, VA 22311-1772

8. Dr. Richard L. Wexelblat 1
   543 Prescott Road
   Merion Station, PA 19066

9. Dr. Bernard Moulin 1
Département d'informatique
Université Laval
Ste-Foy
Québec, Canada G1K 7P4

10. Professor M. Sergot 1
Department of Computing
Imperial College
180 Queen's Gate
London SW7 2BZ
UNITED KINGDOM

11. Professor Edgar Sibley 1
Department of Information and Software Engineering
George Mason University
Mail Stop 4A4
Fairfax, VA 22030-4444

12. Professor Morris Sloman 1
Department of Computing
Imperial College
180 Queen's Gate
London SW7 2BZ
UNITED KINGDOM

13. Mr. John Custy 1
SPAWAR Systems Center
Code D4521
53560 Hull Street, Bldg. C60, Rm 207
San Diego, CA 92152-5800

14. Dr. Jonathan Moffett 1
Department of Computer Science
University of York
York Y01D 5DD
UNITED KINGDOM

15. Language Technology Group 1
2 Buccleuch Place
Edinburgh EH8 9LW
UNITED KINGDOM